

版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF

- Spring Boot微服务构建和微服务无缝接入AWS云平台
- 微服务开发、设计，自动化扩缩容，微服务监控
- 依托Docker容器，在容器中部署微服务
- Docker+Marathon+Mesos集群管理



Spring微服务

Spring Microservices

(印) 拉杰什·RV ©著
文彦峰 彭艳飞©译



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

[PACKT]
PUBLISHING

Spring 微服务

(印) 拉杰什. RV 著

文彦峰 彭艳飞 译

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

Spring 是一个基于 Java 平台的应用程序框架, 基于 Spring 的开发基本已经成为业界的一种规范。本书将一步一步告诉你如何使用 Spring 来开发微服务, 并深度学习 Spring Boot、Spring Cloud、Docker、Mesos 和 Marathon 各个主流框架的使用方法。书中的案例都是基于最新的 Spring 框架所编, 这样你会学习如何编写一个最新潮、最稳定的基于 Java 语言的系统。

本书适用于高等学校计算机及相关专业的教师、学生, 以及 Spring 开发人员、架构师等技术工作人员。

Copyright ©Packt Publishing 2016. First published in the English language under the title ‘Spring Microservices-(9781786466686)’.

本书简体中文版专有翻译出版权由 Packt Publishing 授予电子工业出版社。

版权贸易合同登记号 图字: 01-2017-5386

图书在版编目 (CIP) 数据

Spring 微服务 / (印) 拉杰什·RV (Rajesh RV) 著; 文彦峰, 彭艳飞译. —北京: 电子工业出版社, 2018.6

书名原文: Spring Microservices

ISBN 978-7-121-34085-7

I. ①S… II. ①拉… ②文… ③彭… III. ①互联网络—网络服务器 IV. ①TP368.5

中国版本图书馆 CIP 数据核字 (2018) 第 077241 号

策划编辑: 董亚峰

特约编辑: 穆丽丽

责任编辑: 刘小琳

印 刷: 三河市鑫金马印装有限公司

装 订: 三河市鑫金马印装有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 720×1 000 1/16 印张: 21.5 字数: 420 千字

版 次: 2018 年 6 月第 1 版

印 次: 2018 年 6 月第 1 次印刷

定 价: 88.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式: liuxl@phei.com.cn, (010) 88254538。

Spring 微服务之我见

文彦峰 彭艳飞

微服务正被越来越多的组织和团体关注。微服务架构有很多好处，它通过分解巨大单体式应用为多个服务方法解决复杂性问题。在功能不变的情况下，应用被分解为多个可管理的分支或服务。每个服务都有一个 API 定义清楚的边界。微服务架构模式给采用单体式编码方式很难实现的功能提供模块化的解决方案。由此，单个服务很容易开发、理解和维护。这种架构使每个服务都可以由专门开发团队来开发，并且每个微服务都可以独立部署。

但是，微服务架构也有它的不足。微服务是一个分布式的系统架构，由此带来固有的复杂性，开发者需要了解 RPC 或者 RESTful 接口之间的消息传递内容，甚至协议。如果是单体式开发，开发者就不需要过多地关注接口，只需要关注其本身业务内容的开发即可。

在微服务架构应用中，需要更新不同服务使用的数据库，甚至是不同的数据库类型。这就需要我们了解分布式事务，或者定义一个最终一致的方法，从而对开发者提出更高的要求和挑战。

另外一个挑战在于，微服务架构模式应用的改变将会波及多个服务。例如，完成一个案例，需要修改服务 A、B、C，而 A 依赖 B，B 依赖 C。在单体式应用中，只需要改变相关模块，整合变化，然后部署就可以了。相比之下，微服务架构模式需要考虑相关改变对不同服务的影响。

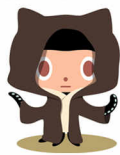
测试一个基于微服务架构的应用也是很复杂的任务。例如，测试一个单体式 Web 应用的 REST API 是很容易的事情；反之，采用流行的 Spring Boot 架构，

同样的服务测试需要启动与其有关的所有服务，这样就给测试带来了一定的复杂性。

部署一个微服务应用也很复杂，而一个单体式应用只要简单地在网关后面部署各自的服务即可。相比之下，一个微服务应用一般由大批服务构成，如果它们互相依赖，就需要全部部署起来，这样才能通信并完结一个业务流程。此外，还需要一个服务发现机制，用来发现与其进行通信服务的地址。

本书由浅入深的介绍了基于 Spring Boot 的微服务开发，可以帮助读者了解 Spring Boot、Spring Cloud、Docker、Mesos 和 Marathon 的使用，详细介绍了 Spring Cloud 各种能力的实现。同时也讲述了微服务的自动化扩（缩）容，以及服务的日志记录和微服务的监控。读完本书，你能够很容易的搭建一套基于 Spring Boot 的微服务系统。

本书的翻译经历了很多困难，能够顺利完成，需要特别感谢吴疆、刘子豪、宋达彬、陈灿、叶东林、张凯旋、何文雅、陈旭泉、吴以林、颜金玉等人的大力协助。



前言



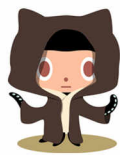
微服务是一种架构风格和模式，通过将复杂系统分解成更小的、彼此协同工作的服务，形成大规模的商业服务。微服务是自主的、自包含的和可独立部署的服务。在当今世界，许多企业构建大型的、面向服务的企业应用程序时，都默认将微服务作为标准。

Spring 框架是多年以来开发社区流行的编程框架。Spring Boot 取消了重量级的应用容器，并提供了轻量级部署——Serverless 架构应用。Spring Cloud 结合了许多 Netflix OSS 组件，并提供了一个生态系统来运行和管理大型微服务。它提供了负载均衡、服务注册、服务监控、服务网关等能力。

然而，微服务也面临着许多挑战，特别是在大规模部署的时候，如监控、管理、分发、扩容、服务发现等。单纯使用微服务而不解决这些常见的微服务问题，会导致灾难性的后果。这本书最重要的部分是一个与技术无关的微服务能力模型，有助于解决常见的微服务挑战。

本书的目标是以务实的方式指导读者实现大规模实施响应式微服务，帮助读者深入了解 Spring Boot、Spring Cloud、Docker、Mesos 和 Marathon。读者将会理解 Spring Boot 如何通过去除重量级应用服务器，来实现自主、无须服务器的部署。读者将学习 Spring Cloud 各种能力的实现，集装箱化的 Docker、Mesos 和 Marathon 的使用，并学会抽象计算资源和控制集群范围。

我相信你会喜欢这本书的每一章节。本书通过成功构思微服务，可以为你的生意增加巨大的价值。本书通过一些案例来体现微服务的实践能力，包括一个旅游领域的研究案例。最后，你将学会如何使用 Spring 框架、Spring Boot 和 Spring



Cloud 实现微服务的体系结构。它们是经过测试的、具有强大功能的工具，可以开发和部署可扩展的微服务。在本书的帮助下，你可以使用 Spring 的最新规范来构建现代的、互联网规模的 Java 应用程序。

章节概要

第 1 章“解密微服务”，涵盖了微服务的基本概念、演变过程，以及它们与面向服务的架构的关系，并且介绍了云原生和十二要素应用程序的概念。

第 2 章“用 Spring Boot 构建微服务”，介绍了使用 Spring 框架构建 REST 和基于消息的微服务，以及如何用 Spring Boot 包装它们。另外，我们还将探索 Spring Boot 的一些核心功能。

第 3 章“微服务概念的应用”，通过详细描述开发人员在企业级微服务中所面临的挑战，来阐述微服务的实践性，并且会总结成功管理微服务生态系统所需的能力。

第 4 章“微服务的演变——一个案例的学习”，通过研究 BrownField 航空公司的微服务演变案例，向读者展示如何应用前面章节所学的微服务概念。

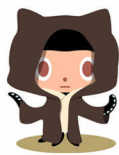
第 5 章“通过 Spring Cloud 对微服务进行扩（缩）容”，展示了如何使用 Spring Cloud 堆栈功能扩容之前构建的微服务。本章详细介绍了 Spring Cloud 的架构和组件，以及它们是如何整合在一起的。

第 6 章“自动化扩（缩）容微服务”，演示了使用服务网关和简单的生命周期管理器，来实现微服务的弹性和自我管理。本章向读者展示了在现实世界中，可以使服务网关更加智能。

第 7 章“日志记录和监控微服务”，涵盖了日志在开发和监控微服务中的重要性。在这里，我们将详细介绍使用开源工具进行集中式日志记录和监控的最佳实践，以及如何将它们与 Spring 项目集成。

第 8 章“用 Docker 实现容器化微服务”，解释容器化在微服务环境中的概念。本章通过 Mesos 和 Marathon 演示了如何大规模部署微服务，来替换自定义生命周期管理器的实现方式。

第 9 章“使用 Mesos 和 Marathon 管理 Dockerized 微服务”，介绍了微服务的



自动配置和部署。本章你也将学习如何使用 Docker 容器将前面构建的微服务进行大规模部署。

准备工作

第 2 章“用 Spring Boot 构建微服务”，需要下列软件来运行代码：

- JDK 1.8
- Spring Tool Suite 3.7.2 (STS)
- Maven 3.3.1
- Spring Framework 4.2.6.RELEASE
- Spring Boot 1.3.5.RELEASE
- Spring-Boot-cli-1.3.5.RELEASE-bin.zip
- RabbitMQ 3.5.6
- FakeSMTP

第 5 章“通过 Spring Cloud 对微服务进行扩（缩）容”，除了上述软件之外还需要：

- Spring Cloud Brixton.RELEASE

在第 7 章“日志记录和监控微服务”中，我们将介绍集中式日志如何在微服务中实现，这需要以下软件堆栈：

- Elasticsearch 1.5.2
- Kibana-4.0.2-darwin-x64
- Logstash 2.1.2

第 8 章“用 Docker 实现容器化微服务”将演示如何使用 Docker 进行微服务部署，这需要以下软件组件：

- Docker version 1.10.1
- Docker Hub

第 9 章“使用 Mesos 和 Marathon 管理 Dockerized 微服务”使用 Mesos 和 Marathon 将 Dockerized 微服务部署到可自动扩展的云中。需要使用下列软件组件：



- Mesos version 0.27.1
- Docker version 1.6.2
- Marathon version 0.15.3

读者对象

本书主要针对 Spring 开发人员，他们希望构建互联网规模应用程序以满足现代业务需求。本书通过检验一些真实的用例和实践代码实例，来帮助开发人员了解究竟什么是微服务，以及当今社会微服务为什么如此重要。开发人员将懂得如何构建简单的 RESTful 服务，并将其有机地发展成为真正的企业级微服务生态系统。

对架构师而言，他们使用 Spring 框架、Spring Boot、Spring Cloud 设计强大的互联网规模微服务，并用 Docker、Mesos 与 Marathon 进行管理。当他们需要在这方面寻求帮助时，本书中的能力模型将帮助架构师设计出超出本书所讨论的工具和技术的方案。

约定

在本书中，你会发现很多区分多种不同信息的文本样式。这里是一些样式的例子及其解释。

文本中的代码、数据库表名、文件夹名称、文件名、文件扩展名、路径名、虚拟 URLs、用户输入和 Twitter 句柄等如下所示：

“下列属性可以在 `application.properties` 进行自定义应用程序相关信息。”

代码块设置如下：

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.3.4.RELEASE</version>
</parent>
```



任何命令行输入或输出的写法如下：

```
$ java -jar fakeSMTP-2.0.jar
```

提示和技巧显示如下：

提示

使用 AWS Lambda，研发人员可以将应用托管到云服务平台。

下载代码文件

本书的代码文件托管在 GitHub 上，你可以通过 <https://github.com/369945969/Spring-Microservices> 下载代码文件。

文件下载完成后，请确保使用正确的解压缩工具：

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

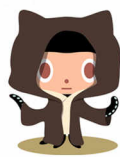


目 录

第 1 章 解密微服务	1
微服务的演进	2
命令式架构的演进	4
什么是微服务	5
微服务——蜂窝类比	8
微服务原则	8
微服务的特性	10
微服务中服务的特性	11
微服务案例	17
微服务的好处	23
与其他架构风格的联系	32
微服务使用案例	41
总结	45
第 2 章 用 Spring Boot 构建微服务	46
设置开发环境	47
开发 RESTful 服务——传统方法	47
传统 Web 应用转移到微服务	51
使用 Spring Boot 构建 RESTful 微服务	51
开始使用 Spring Boot	52
使用 CLI 开发 Spring Boot 微服务	53
使用 STS 开发 Spring Boot Java 微服务	54
下一步是什么	65



Spring Boot 配置	65
修改默认嵌入的 Web 服务器	68
实现 Spring Boot 安全性	69
为微服务开启跨域访问	73
实现 Spring Boot 通知	74
Spring Boot Actuator	87
配置应用信息	89
添加自定义运行状况模块	89
记录微服务	92
总结	93
第 3 章 微服务概念的应用	94
模式和常见设计决策	95
微服务的挑战	126
微服务能力模型	131
总结	136
第 4 章 微服务的演变——一个案例的学习	137
回顾微服务能力模型	138
理解 PSS 应用	139
庞然大物的终结	143
使用微服务来拯救	149
业务用例	150
为演化制订计划	150
只有在需要时迁移模块	168
目标架构	168
目标实现视图	174
总结	179
第 5 章 通过 Spring Cloud 对微服务进行扩（缩）容	180
回顾微服务	181
回顾 BrownField 航空的 PSS 系统实践	182
什么是 Spring Cloud	182
建立 BrownField PSS 的环境	187
Spring Cloud Config	187



一个声明式的 REST 客户端 Feign	202
用于负载均衡的 Ribbon	204
注册和发现的 Eureka	206
API 网关——Zuul 代理	216
反应式微服务流	224
BrownFeild PSS 架构总结	228
总结	229
第 6 章 自动化扩（缩）容微服务	230
回顾微服务功能模型	230
用 Spring Cloud 扩（缩）容微服务	231
理解自动化扩（缩）容的概念	233
自动化扩（缩）容方法	238
总结	250
第 7 章 日志记录和监控微服务	251
回顾微服务能力模型	252
理解日志管理的挑战	253
集中式日志解决方案	254
日志方案的选择	256
微服务监控	265
使用数据湖泊的数据分析	276
总结	277
第 8 章 用 Docker 实现容器化微服务	279
回顾微服务功能模型	280
理解 BrownField PSS 微服务的区别	281
什么是容器	282
VMs 与容器之间的区别	283
容器的好处	284
微服务和容器	286
Docker 简介	286
在 Docker 中部署微服务	290
在 Docker 上运行 RabbitMQ	294
使用 Docker Registry	294

云上的微服务·····	295
在 EC2 上运行 BrownField 服务·····	296
更新生命周期管理器·····	297
容器化的未来——内核和强化安全·····	298
总结·····	298
第 9 章 使用 Mesos 和 Marathon 管理 Dockerized 微服务 ·····	299
回顾微服务功能模型·····	300
缺少的部分·····	300
为什么集群管理很重要·····	301
集群管理能做什么·····	302
与微服务的关系·····	305
与虚拟化的关系·····	305
集群管理解决方案·····	305
集群管理与 Mesos 和 Marathon·····	309
为 BrownField 微服务实现 Mesos 和 Marathon·····	313
生命周期管理器的部署·····	325
技术元模型·····	326
总结·····	327

第 1 章

解密微服务



微服务是用于满足现代业务需求的一种架构风格和软件开发方法。微服务不是一种发明，它更多的是之前架构风格的一种演进。

我们首先会深入探索从传统单机架构演进而来的微服务架构，之后介绍微服务的定义、概念和特性。最后，我们会分析微服务的经典案例，比较微服务和其他面向服务架构（SOA）及十二因素应用架构方法的相同点和关系。十二因素应用定义了一套在云上开发应用程序的软件设计原则。

在这一章，您会学到：

- 微服务的演进。
- 微服务架构定义和示例。
- 微服务架构的概念和特性。
- 微服务架构的经典实用案例。
- 微服务和 SOA 及十二因素应用的联系。

微服务的演进

微服务是继 SOA 之后日益流行的架构模式之一，由 DevOps 和云提供补充。这一节会详细说明，近几年现代企业中颠覆性的创新趋势和技术演进是促进微服务发展的两个因素。

微服务演进的催化剂之业务需求

在这个向数字化转型的时代，企业更多地采用技术革新来增加收益和扩大客户群。企业主要使用社交媒体、手机、云、大数据和物联网来实现颠覆性创新。利用这些技术，企业找到了快速渗透市场的新途径，这给传统 IT 交付机制带来了严重挑战。

传统开发和微服务在敏捷性、交付速度和扩展性等方面的对比如图 1-1 所示。

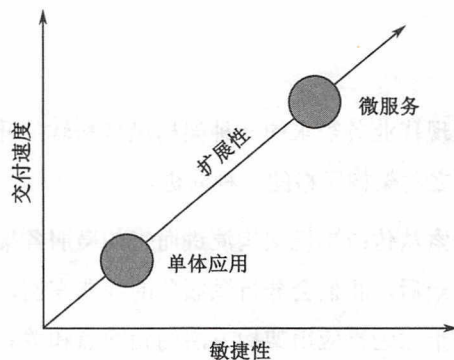


图 1-1

提示

与传统的单体应用相比，微服务更灵活，并且可以实现更快的交付速度和更大的规模部署。

在几年的转折中，商业投资大型应用程序开发的时代已经过去了。企业不再像几年前一样通过综合应用程序来管理其端到端业务功能。图 1-2 所示的成本关系图

显示了传统单体应用和微服务的周转时间和成本差异。

提示

微服务提供开发快速敏捷应用程序的方法，从而降低总体成本。

如今，航空公司和金融机构不会重建他们复杂而庞大的核心主架构系统，零售商和其他行业也不会重构重量级的供应链管理应用程序，如其传统的 ERP。焦点已转移到以最敏捷的方式来满足企业的特定需求。我们举一个使用传统的单体应用程序运行的在线零售商的例子，如果零售商想基于顾客之前的购物偏好为其提供个性化产品，或依据其购买倾向来促销产品等，他们将很快建立一个个性化引擎，或基于目前的需求为当前应用程序引入插件。

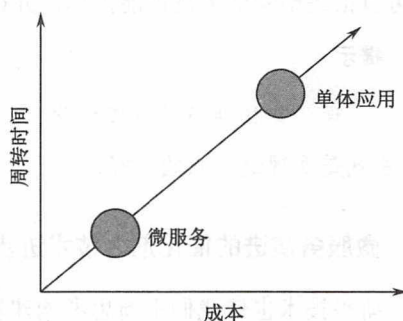


图 1-2

同图 1-3 (a) 一样，我们通常是对业务系统进行改造，进行新功能的开发，而不是去重写系统的核心功能；或如图 1-3 (b) 所示，修改核心系统的功能去调用一些新的功能，而这些功能通常称为微服务。

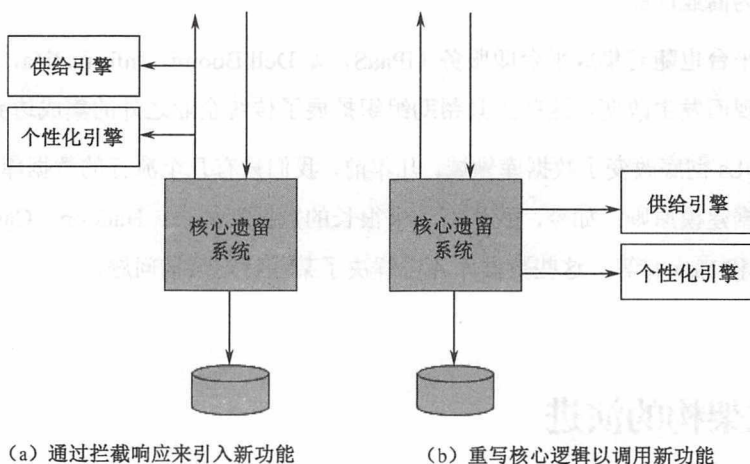


图 1-3

这种方法给团队提供了大量的机会，即可以以较低的成本快速尝试新功能。企

业可以很快地验证关键性能指标，并在需要时更改或替换这些实现。

提示

现代架构期望尽量模块化，并且最小化成本来替换这些模块。微服务方法就是实现这一点的途径。

微服务演进的催化剂之技术进步

新兴技术也使我们重新思考构建软件系统的方式。例如，几十年前，我们甚至不能想象一个没有两阶段提交的分布式应用。现在，NoSQL 数据库使我们以不同的方式来思考。

类似地，这些技术上的范式转移重塑了软件架构的所有层面。

HTML5 和 CSS3 的出现及移动应用程序的进步重新定位了用户界面。客户端 JavaScript 框架，如 Angular、Ember、React、BackBone 等，由于它们的客户端渲染和响应式设计而变得极其流行。

随着云计算变成主流，平台即服务（PaaS）供应商，如 Pivotal CF、AWS、Salesforce.com、IBMs Bluemix、RedHat OpenShift 等使我们不得不重新评估构建中间件的方式。由 Docker 引发的容器革命极大地影响了基础设施领域。现在，基础设施已经被视为商业产品。

集成平台也随着集成平台即服务（iPaaS，如 Dell Boomi, Informatica, MuleSoft 等）的出现而发生改变。这些工具帮助组织扩展了传统企业之外的集成边界。

NoSQLs 彻底改变了数据库领域。几年前，我们只有几个流行的数据库，都是基于传统数据建模原则；如今，我们有一个很长的数据库列表：Hadoop、Cassandra、CouchDB 和 Neo 4j 等。这些数据库各自解决了某些特定架构问题。

命令式架构的演进

应用程序架构一直在满足苛刻的业务需求和技术的演变。架构经历了从古老的大型机系统发展到完全抽象的云服务的过程，如 AWS Lambda。

提示

使用 AWS Lambda，研发人员可以将应用托管到云服务平台。

获取更多关于 Lambda 的信息请访问：<https://aws.amazon.com/documentation/lambda/>。

不同的架构方法和风格，如大型主机、客户服务器、N-层、面向服务，在不同时期都很流行。无论选择哪一种架构风格，本质上都是单机架构。微服务架构是现代需求演变的结果，如敏捷、快速交付、新兴技术和从前几代架构吸取的经验。

微服务帮助我们打破单体应用的边界，并在一个系统中构建逻辑独立的小型系统，如图 1-4 所示。

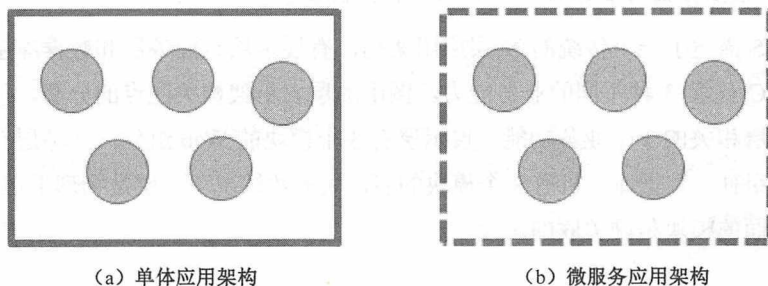


图 1-4

提示

如果我们把单体应用看作一组包含物理边界的逻辑子系统，那么微服务是一组没有封闭物理边界的独立子系统。

什么是微服务

微服务是当今许多软件应用领域使用的一种架构风格，微服务以其实现高度敏捷、快速交付及可扩展性而闻名。微服务给我们提供了一种物理隔离的模块化应用的开发方式。

微服务不是一个新的发明。许多组织，如 Netflix、Amazon 和 eBay，成功地分而治之，从功能上将它们的应用划分成更小的原子单元，各自执行单一的功能。这些组织解决了许多他们在单体应用上碰到的普遍问题。

随着这些组织的成功，其他组织开始采取这种模式作为一种常规模型来重构他们的单体应用。之后，推广者把这个模式称为微服务架构。

微服务最初起源于 Alistair Cockburn 创造的六边形架构的想法。

提示

阅读更多关于六边形架构请访问：<http://alistair.cockburn.us/Hexagonal+architecture>。

微服务是一种架构风格或者是一种构建一套具备业务能力的 IT 系统的方法，这些业务能力包括自治、自包含和松耦合。

图 1-5 描述了一个传统的 N-层应用架构，有展示层、业务层和数据库层。模块 A、B 和 C 代表 3 种不同的业务能力。图中的层表示架构关注点的分离。每一层都拥有与该层相关的 3 个业务功能。展示层有 3 个模块的 Web 组件，业务层有 3 个模块的业务组件，数据库主机有 3 个模块的表。大多数情况下，层是物理上可扩展的，注意层里面的模块是硬关联的。

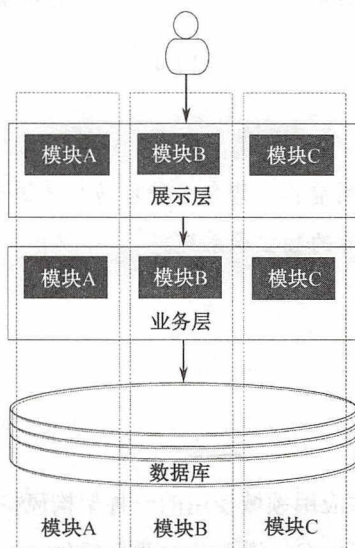


图 1-5

现在我们来查看一个基于微服务的架构。

我们可以在图 1-6 中注意到，在微服务中边界是反向的。每一个垂直分片代表一个微服务。每一个微服务有属于它自己的展示层、业务层、数据库层。微服务与业务能力保持一致。通过这样做，改变一个微服务并不影响其他的微服务。微服务并没有一个标准的通信和传输机制。通常，微服务之间的互相通信广泛采用轻量级协议，如 HTTP 和 REST；或者消息协议，如 JMS 和 AMQP。特殊情况下，可能会选择更加优化的通信协议，如 Thrift、ZeroMQ、Protocol Buffers 或 Avro。

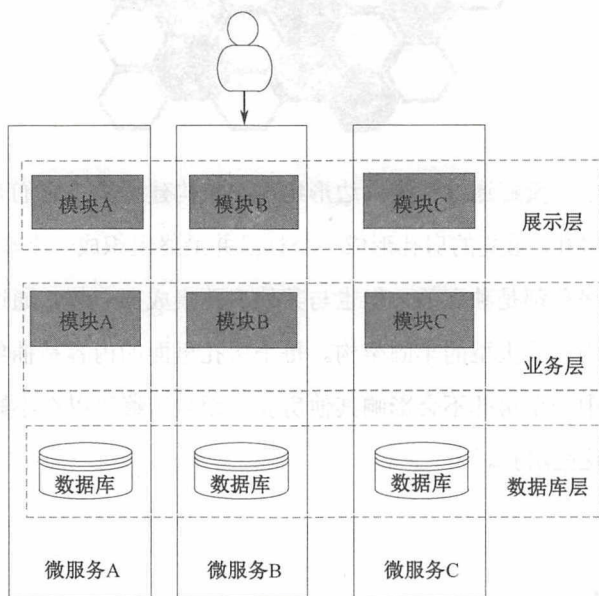


图 1-6

由于微服务更符合业务功能，并且有可独立管理的生命周期，因此它是企业开展 DevOps 和云的理想选择。

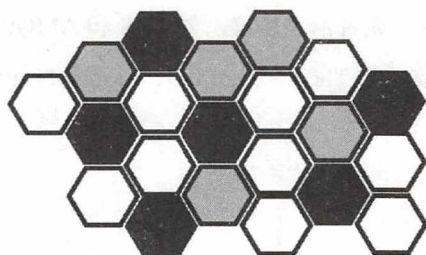
提示

DevOps 是促进 IT 开发和运维之间的合作，从而获得更高的效率。

获取更多关于 DevOps 的信息可见：<http://dev2ops.org/2010/02/what-is-devops/>。

微服务——蜂窝类比

蜂窝是微服务架构的一个理想的类比。



在现实世界中，蜜蜂通过标准六边形蜡房孔来构建蜂巢。它们开始很小，用不同的材料来构建房孔。重复的房孔形成一个样品并最终组织成一个牢固的组织结构。蜂巢中的每一个房孔都是独立的，但也与其他房孔集成在一起。通过新增房孔，蜂巢有组织地成长为一个大型的牢固架构。每个房孔里面的内容是抽象的，而且对外不可见。损坏其中一个房孔不会影响其他房孔，而且蜜蜂可以在不影响整体蜂巢的情况下重新构建这些房孔。

微服务原则

本节我们会考察微服务架构的一些原则。这些原则是设计和开发微服务时的“必需品”。

服务的单一职责

单职责原则是 SOLID 设计模式定义的原则中的一部分，它表述的是一个单元模块只应该有一个职责。

提示

获取更多关于 SOLID 设计模式可见：<http://wiki.c2.com/?PrinciplesOfObjectOrientedDesign>。

这表明，一个单元，可以是一个类，一个方法，或者是一个服务；应该只有一个职责，任何时候不应该有两个单元模块分担一个职责或者一个单元模块有不止一个职责。一个单元模块有不止一个职责时意味着高耦合。

如图 1-7 所示，客户、产品和订单在电子商务应用中是不同的功能。与其将它们构建在一个应用中，不如用 3 个不同的服务，每个恰好负责一个业务功能，所以更改某个功能职责时不会影响其他的。在图 1-7 情景中，客户、产品和订单被处理为 3 个独立的微服务。

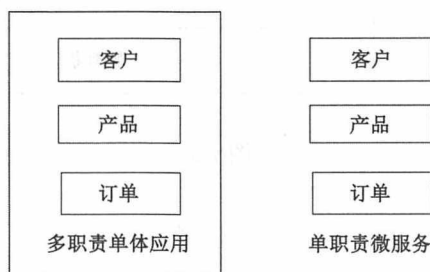


图 1-7

微服务是自治的

微服务是自包含的、可独立部署的和自治的服务，它全面负责业务的功能和执行。它们绑定所有的依赖，包括包依赖和运行环境，如 Web 服务器和容器或者虚拟机之类的抽象物理资源。

微服务和 SOA 之间的一个主要不同点在于它们的自治层级。大多数 SOA 应用提供服务级的抽象，微服务更进一步抽象了实现和运行环境。

在传统应用开发中，我们构建一个 WAR 或者 EAR，然后将它部署到一个 JEE 应用服务器，如 Jboss、WebLogic、WebSphere 等。我们可能在同一个 JEE 容器中部

署多个应用。在微服务中，每个微服务会被构建成一个 Jar 包，嵌入所有的依赖，以一个独立的 Java 进程来运行。

微服务也可能由它自己的容器来执行，如图 1-8 所示。容器是便携、独立管理和轻量级的运行环境。容器技术，如 Docker，是部署微服务的理想选择。

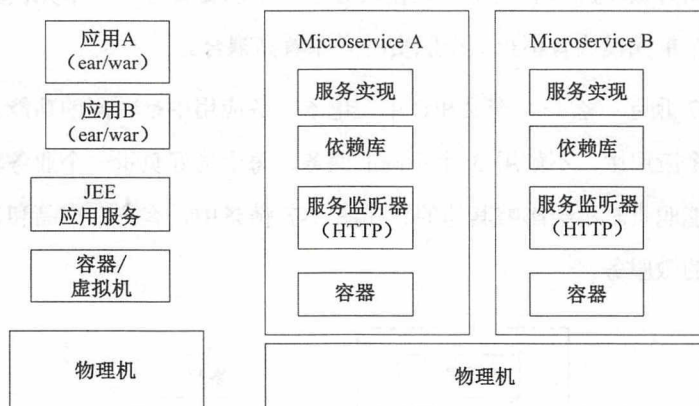


图 1-8

微服务的特性

本章前面关于微服务的定义是不严谨的。理论者和实践者对微服务有很强烈但有时不一样的观点。微服务还没有一个单一的、具体的和普遍被接受的定义。然而，所有成功的微服务实现展示出许多共同的特性。因此，理解这些特性比执着于理论上的定义更重要。本节会详细说明所有成功的微服务之间的共同特性。

服务是一等公民

在微服务的世界中，服务是一等公民。微服务以 API 的形式暴露服务端口，并抽象所有的实现细节。内部实现逻辑、架构和技术（包括编程语言、数据库和服务机制的质量等）是完全隐藏在服务 API 后面的。

而且，在微服务架构中没有更多的应用开发，与之替代的是组织者关注于服务开发。在大多数企业中，需要在构建应用程序的方式上发生重大思想转变。

在一个用户微服务中，内部的数据结构、技术、业务逻辑等都是隐藏的，它们不会暴露或对外部实体可见。例如，用户微服务可能会暴露注册用户和获取用户两个 API 供他人调用。

微服务中服务的特性

微服务或多或少有一点 SOA 的风味，许多在 SOA 中定义的服务特性也适用于微服务。

如下所示：

- **服务约定：**类似于 SOA，微服务是通过定义好的服务约定来描述的。在微服务的世界中，JSON 和 REST 是被普遍接受用于服务通信的。在 JSON/REST 的用例中，有许多技术用于定义服务约定。例如，JSON Schema、WADL、Swagger 和 RAML 等。
- **松散耦合：**微服务是独立和松耦合的。在大多数用例中，微服务接收一个事件输入并响应另外一个事件。Messaging，HTTP 和 REST 在微服务交互中被广泛使用。基于消息的端提供更高层次的解耦。
- **服务抽象：**在微服务中，服务抽象不仅仅是服务实现的抽象，也给所有库和环境明细提供一个全面的抽象。
- **服务重用：**微服务是粗粒度的可重用业务服务。可以被移动设备、桌面、其他微服务甚至其他系统访问。
- **无状态：**微服务设计是无状态的，而且没有任何共享状态或服务维持的会话状态。这里的用例需要维持状态，它们被维持在数据库或内存中。

- **服务是可发现的：**微服务是可发现的。在一个经典微服务环境中，微服务自我告知它们的存在并且它们自己可被发现。当服务停止，它们自动从微服务体系中脱离出来。
- **服务互通性：**服务是互通的，因为它们使用标准协议和消息交互标准。Messaging, HTTP 等用作传输机制。在微服务世界中，REST/JSON 是微服务中使用最为流行的协议。如果需要在通信上进一步优化，可以用其他的协议，如 Protocol Buffers、Thrift、Avro 或 Zero MQ。不过，使用这些协议可能会限制服务的整体互通性。
- **服务可组合性：**微服务是可以组合的。服务可组合性是通过服务匹配或服务编排来完成的。

提示

获取更详细的 SOA 原则可见：<http://serviceorientation.com/serviceorientation/index>。

微服务是轻量级的

设计良好的微服务趋向于单个业务能力的实现，所以它们只执行一个功能。因此，我们在大多数实现中看到微服务的一个共同特征就是规模比较小。

当选择支持的技术，如 Web 容器，我们会保证它也是轻量级的以至于整体规模依旧易于管理。例如，相比于更加复杂和传统的应用容器，如 WebLogic、WebSphere、Jetty 和 Tomcat 是作为应用容器更好的选择。

相比于管理程序，相比 VMWare 和 Hyper-V，Docker 的容器技术也帮助我们保持尽量小的基础规模。

如图 1-9 所示，微服务通常部署在 Docker 容器中，它封装了业务逻辑和所需要的库。帮助我们快速复制整体配置到一个新的机器或者一个完全不同的主机环境，甚至移动到一个不同的云环境。由于没有物理基础设施依赖，容器化的微服务易于移植。

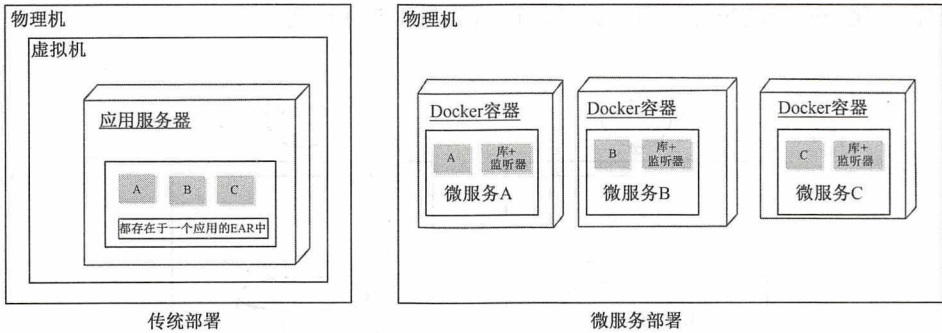


图 1-9

多语言架构的微服务

由于微服务是自治的，抽象了服务 API 背后的一切，使不同的微服务有不同的架构成为可能。我们在微服务实现中看到的一些共同特性是：

- 不同服务使用相同技术的不同版本。一个微服务可能是在 Java1.7 上编写的，另一个可以在 Java1.8 上编写。
- 用不同语言开发不同微服务。例如，一个微服务用 Java 开发，另一个用 Scala 开发。
- 使用不同架构，如一个微服务用 Redis 缓存服务数据，而另一个微服务用 MySQL 作为持久化数据存储。

在图 1-10 的例子中，作为 Hotel Search 期望有高交易量，且具有严格的性能要求，它是用 Erlang 实现的。为了支持预测搜索，Elastic Search 用来作为数据存储。同时，Hotel Booking 需要更多 ACID 事务特征，因此它用 MySQL 和 Java 来实现。内部实现隐藏在以 HTTP 为基础的 REST/JSON 的服务端之后。

微服务环境中的自动化

大多数微服务实现从开发到生产的最大自动化。

由于微服务将单体应用程序拆分为多个较小的服务，大型企业可能会经历微服务的扩展。除非有非常好的自动化工具，否则大量的微服务很难管理。微服务的更小规模也为开发到部署的过程实现自动化提供了可能。通常，微服务是端到端

的自动化——自动编译、自动测试、自动部署和弹性扩展。

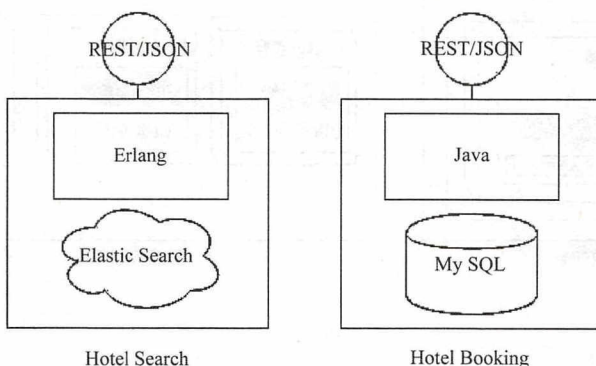


图 1-10

图 1-11 表明，自动化典型应用于开发、测试、发布、部署阶段：



图 1-11

- 开发阶段自动化使用版本控制工具，如 Git，结合持续集成（CI）工具，如 Jenkins、Travis CI 等。可能也包括代码质量检测和自动化单元测试。使用微服务也可以实现对每段代码植入的完全构建的自动化。
- 测试阶段会通过测试工具实现自动化，如 Selenium、Cucumber 和其他 AB 测试策略。因为微服务与业务能力保持一致，自动化的测试用例的数量相比于单体应用较少，因此也使得每次编译时的回归测试成为可能。
- 基础设施配置通过容器技术完成，如 Docker，结合配置管理工具，如 Chef 或者 Puppet，以及自动化运维工具，如 Ansible。使用 Spring Cloud、Kubernetes、Mesos 和 Marathon 等工具可以实现自动化部署和微服务的管理。

支持微服务的生态系统

大多数大规模的微服务实现都有一个支持的生态系统。这个生态系统包括 DevOps 处理、集中化日志管理、服务注册、API 网关、大量监控、服务路由和流控制机制。

当支持能力满足需求时，微服务一般能够很好地工作，如图 1-12 所示。

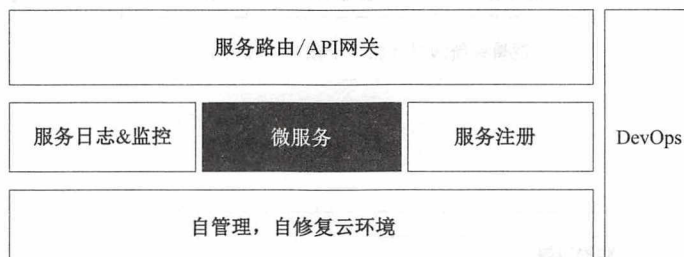


图 1-12

分布式和动态的微服务

成功的微服务在服务中实现逻辑和数据的封装。这导致两个非常规的情景：分布式数据逻辑和分散治理。

相比于将所有逻辑和数据合并到一个程序中的传统应用，微服务分散了数据和逻辑。每个服务，对应于特定的业务能力，拥有其数据和逻辑。

图 1-13 中的虚线表明了逻辑上的单体应用边界。当我们移植这些到微服务时，每个微服务 A、B 和 C 都创建了它自己的物理边界。

微服务通常不像在 SOA 中使用的集中式治理机制。微服务实现中的一个常见特性是它们没有重量级企业产品，如企业服务总线（ESB）。取而代之，业务逻辑和信息被嵌入到服务本身。

一个典型 SOA 应用如图 1-14 所示。购物逻辑完全实现在 ESB 中，对外暴露客户、订单和产品等不同服务。在微服务方法论中，购物自身作为一个独立微服务运行，以松耦合的方式与客户、产品和订单交互。

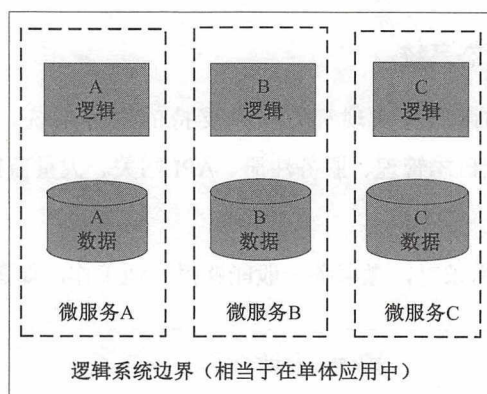


图 1-13

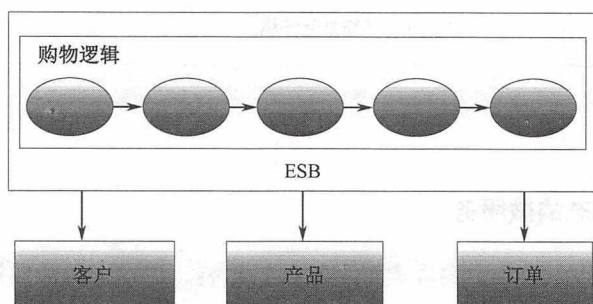


图 1-14

SOA 的实现大量依赖于静态注册和仓库配置，以管理服务和其他组件。而微服务在这上面注入更多动态性质。因此，静态治理方法被视为维持最新信息的瓶颈。这就是为什么大多数微服务实现使用从运行时自动化动态构建注册信息。

反脆弱，快速故障，自我修复

反脆弱是一个在 netflix 上成功尝试的技术。它是在现代软件开发中构建故障安全系统的其中一个最佳方法。

提示

反脆弱概念是 Nassim Nicholas Taleb 在他的书 Antifragile: Things That Gain from Disorder 中介绍的。

在反脆弱的实践中，软件系统始终受到挑战。软件系统通过这些挑战发现不足，进而变得更稳定、更可用。亚马逊的 GameDay 实践和 NetFlix 的 Simian Army 是这些反脆弱实验的很好案例。

快速故障是另外一个概念，用于构建能容错的和能复原的系统。这种哲学理论倡导能预知故障的系统而不是构建不会失败的系统。应重视系统多快会故障，如果发生故障，多快能修复这个故障。使用这种方法，关注点从 Mean Time Between Failures (MTBF) 转移到 Mean Time to Recover (MTTR)。这种方法的核心优势是如果某个服务发生故障，会将自己从集群中剥离，下游功能也不会受到压力。

自我修复在微服务部署中是常用的，系统自动获取故障并自我调整。这些系统也能预防未来故障。

微服务案例

实现微服务时并没有“一刀切”的方法。本节通过分析不同案例来具体化微服务概念。

旅游门户网站案例

在第一个案例中，我们会回顾一个旅游门户网站：Fly By Points。Fly By Points 收集客户通过在线网站预订酒店、航班或者汽车时积累的积分。客户登录到 Fly By Points 网站，他（或她）能看到积分，通过积分兑换可用的个性化优惠，以及可能到来的旅行。

我们假定图 1-15 是登录后的主页。对于 Joe 来说，有 2 个即将到来的旅行，4 个个性化优惠和 21123 个忠诚积分。用户单击每一个方框，会查询并显示出详情。这个旅游门户网站有一个基于 Java Spring 的传统单体应用架构，如图 1-16 所示。

图 1-16 所示的旅游门户网站的架构是基于 Web 且是模块化的，层与层之间有清晰的划分。按照惯例，这个旅游门户网站也是作为单个 WAR 部署在 Web 服务器上的，如 Tomcat。数据存储在一个全面的后台关系型数据库中。当复杂度很低时，这是一个很合适的目标架构。随着业务增长，用户基数扩展，复杂度也会增加。这会

导致交易量的同比增长。这时候，企业应该将这个单体应用重构成微服务，来获取更好的交付速度、敏捷性和可管理性。

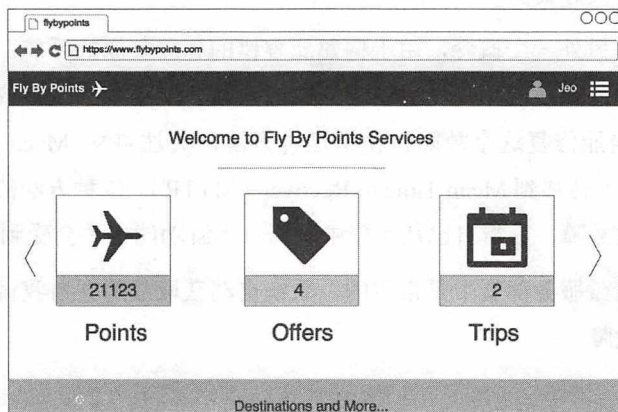


图 1-15

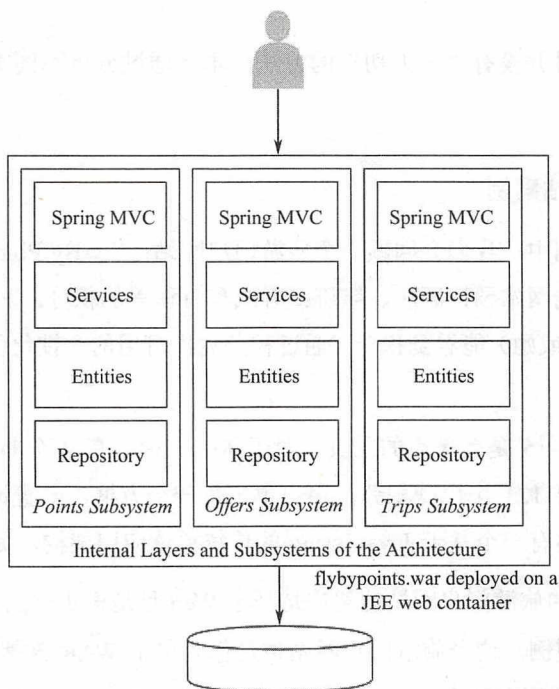


图 1-16

如图 1-17 所示，考察这个应用的简单微服务版本，我们能立即注意到这个架构中的一些东西：

- 每个子系统现在已经成为一个独立的系统，一个微服务。这里有 3 个微服务代表 3 个业务功能：旅行、优惠和积分。每一个都有它自己的内部数据存储和中间层，每一个服务的内部结构相同。
- 每一个服务封装它自己的数据库和它自己的 HTTP 监听器，如 Jetty、Tomcat 等。与之前的模型相反的是，没有 Web 服务器或者 WAR。
- 每一个微服务暴露一个 REST 服务来操作属于这个服务的资源和实体。

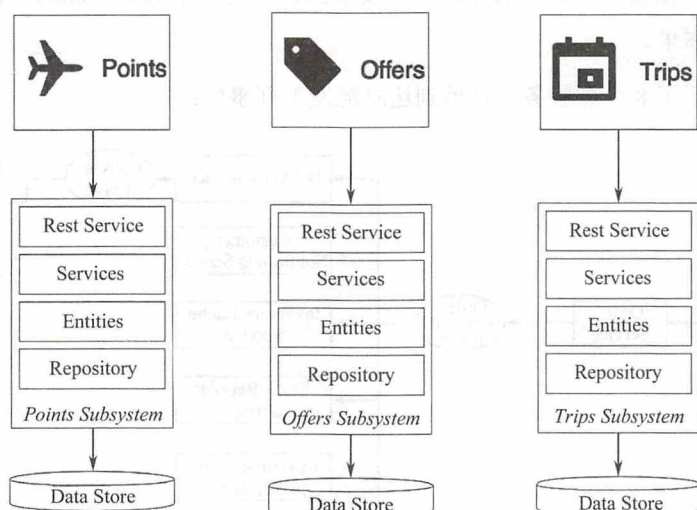


图 1-17

假设展现层使用客户端 JavaScript MVC 框架开发，如 Angular JS。这些客户端框架能直接调用 REST 接口。

当 Web 页面被加载，所有的 3 个方框，Trips、Offers 和 Points 会显示详情，如积分、优惠数量和旅程数量。这是通过每个方框使用 REST 独自发起异步调用给各自的后端微服务来完成的。在服务层的服务之间没有依赖。当用户单击某个方框，屏幕将会跳转并加载被单击方框条目的详细内容，这些都是通过相应的微服务调用来完成的。

基于微服务的订单管理系统

让我们再看另一个微服务案例：一个在线零售网站。在这个例子中，我们会更多关注在后端服务上，如处理客户通过网站下单产生的订单服务：

如图 1-18 所示，这个微服务系统是完全基于 Reactive 编程实践设计的。

提示

更多关于 Reactive 编程请访问：<http://www.reactivemanifesto.org/>。

发布一个事件时，多个微服务准备好在接收到事件时启动。它们中每一个都是独立的，且不依赖于其他微服务。这一模型的优点是可以继续添加或替换微服务以实现特定的需求。

下图显示了 8 个微服务，订单到达时触发下列事件：

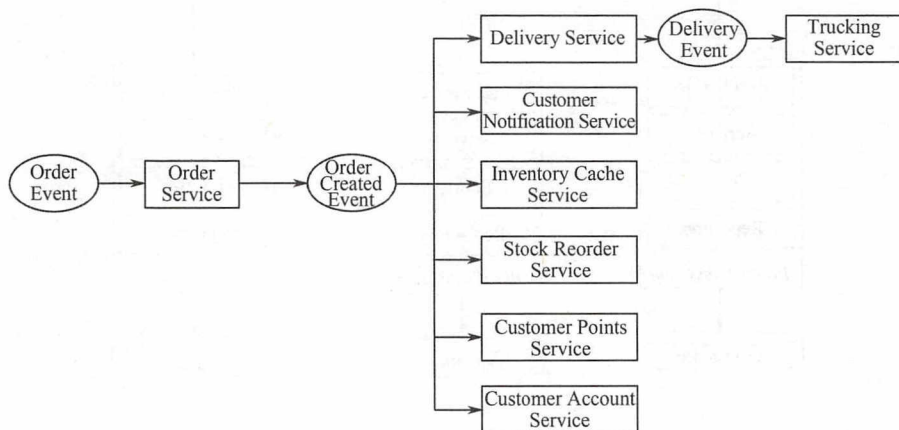


图 1-18

(1) 当接收到订单事件时启动订单服务。订单服务创建一个订单并保存详细信息至其自己的数据库。

(2) 如果订单被成功保存，订单服务创建并发布订单成功事件。

(3) 当订单成功事件到达，会发生一系列动作。

(4) 派送服务接收到事件并展示派送记录以将订单交付给客户。同时，生成并发布派送事件。

(5) 货运服务拿到派送事件并执行，如创建一个运输计划。

- (6) 客户通知服务发送一个通知给客户，告知客户一个订单被处理。
- (7) 库存缓存服务更新库存缓存的可用产品数量。
- (8) 库存重订服务检查库存限制是否足够，并在需要时生成补货事件。
- (9) 客户积分服务根据此次购买重新计算客户的忠诚度积分。
- (10) 客户账户服务更新客户账户中的历史订单记录。

在这个方法中，每个服务只为一个功能负责。服务接收并生成事件。每个服务是独立的并不被其他服务所知。因此，整个集群可以如蜂窝类比中所提到的那样有机地生长，新的服务可以在需要的时候加进来，增加新的服务不影响已存在的服务。

旅行社门户案例

第3个例子是一个简单的旅行社应用。如图 1-19 所示，在这个例子中，我们既会看到同步 REST 调用，也会看到异步事件。

在这个案例中，该门户仅是一个有多菜单项或链接的容器应用。请求指定页面时，如单击一个菜单或链接，会从指定的微服务中加载。

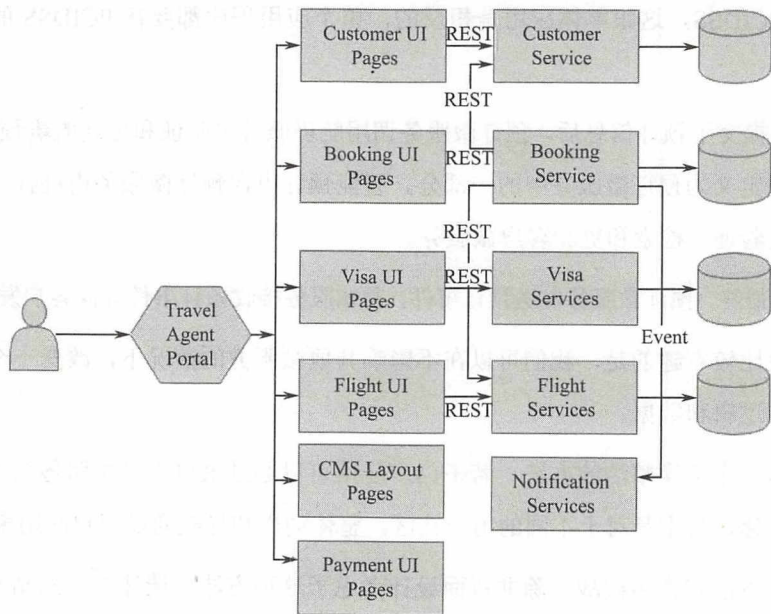


图 1-19

当客户请求预订时，会触发以下事件：

(1) 旅行社打开航班 UI，查找一个航班，并为客户识别正确的航班。在幕后，航班 UI 是从航班微服务中加载的。航班 UI 仅与它后端航班微服务中的 API 进行交互。它通过 REST 请求调用航班微服务来加载航班并显示给用户。

(2) 然后旅行社通过访问客户 UI 来查询客户详情。类似于航班 UI，客户 UI 从客户微服务中加载。客户 UI 上的动作会触发对客户微服务的 REST 调用。客户详情是通过调用客户微服务上的某个 API 来加载的。

(3) 然后，旅行社检查签证明细判断客户是否有去所选国家旅行的资格。模式与前两点相似。

(4) 接下来，和前面一样，旅行社使用从预订微服务中加载出来的预订 UI 创建一个预订单。

(5) 支付页面从支付微服务中加载。通常，支付服务具有如 PCIDSS 承诺（保护或加密动态和静态数据）的额外约束。微服务方法的优点是，任何微服务都不需要考虑 PCIDSS，这跟单体应用是相反的，整个应用程序都要在 PCIDSS 的管理规则下。

(6) 提交了预订信息后，预订微服务调用航班服务来验证和更新航班预订信息。这种编配定义为预订微服务中的一部分。智能预订也在预订微服务内进行，预订流程中，它验证、检索和更新客户微服务。

(7) 最终，预订微服务发送预订事件，通知服务接收预订事件并向客户发送通知。

这里比较有趣的是，我们可以在不影响其他微服务的情况下，改变一个微服务的接口、逻辑和数据。

这是一个干净整洁的方法。若干门户应用可以通过来自不同微服务的不同屏幕组合来构建，特别是对于不同的用户社区。整体动作和导航通过门户应用来控制。

这种方法有许多挑战，除非页面设计考虑了这种方法。请注意，网站布局和静态内容将由内容管理系统（CMS）作为布局模板加载存储在 Web 服务器中。网站布局也可能包含在运行时从微服务加载的 UI 片段。

微服务的好处

与传统的多层单体应用架构相比，微服务具有许多优点。本节介绍了微服务架构方法的一些主要优点。

支持多语言架构

使用微服务，架构师和开发人员可以选择适合各个微服务的体系架构和技术，从而更方便地根据情况得到最佳解决方案。

由于微服务是自治和独立的，每一个微服务可以用其不同版本的架构或技术来运行。

下图是一个简单实用的多语言架构微服务。

有一个审计所有系统交易和记录交易明细的需求，审计内容包括请求和响应数据、启动交易的用户、调用的服务，等等。

如图 1-20 所示，核心服务如订单和产品微服务使用关系型数据存储，审计微服务持久化数据至 Hadoop File System (HDFS)。在存储大量数据（如审计数据）时，关系型数据库既不理想也不划算。在单体应用中，通常使用一个共享的、单一数据库来存储订单、产品和审计数据。

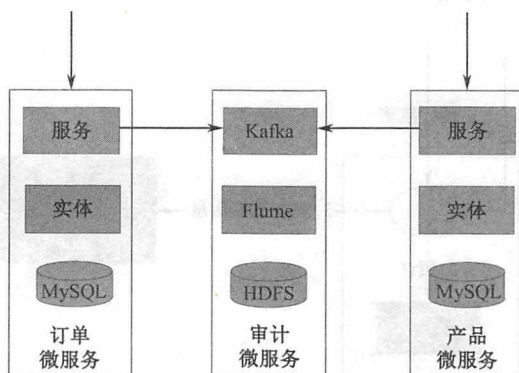


图 1-20

在此例中，审计服务是使用不同架构技术的微服务。同样地，不同功能的服务也可以使用不同的架构。

也有可能预订微服务运行在 Java 7 上，而搜索微服务可能运行在 Java 8 上。同样地，订单微服务可能用 Erlang 编写，派送微服务可能用 Go 语言编写。对于单体架构而言，这些都是不可能的。

能够实验和创新

现代企业正在快速成长。微服务是企业通过提供实验和快速失败能力来进行颠覆性创新的关键推动力之一。

由于服务相当简单，而且规模较小，企业负担得起用于实验新功能、算法和业务逻辑的风险。对于大型单体应用，实验并不容易，也不简单划算。商家需要花大价钱去编译或改变一个应用来尝试一些新的东西。对于微服务，可以编写一个小的微服务去实现目标功能，并以反应型的方式将其嵌入系统。然后可以用几个月的时间来测试新的功能，如果新的微服务没有按预期工作，我们可以用另外一个微服务更换或替代它。相比于单体应用的方法，替换它的代价是相当小的。

在另一个航班预订网站的案例中，航空公司想在它们的预订网页上展示个性化酒店推荐。推荐信息必须显示在预订确认页面。

如图 1-21 所示，相比于合并这个需求至单体应用本身，写一个可以嵌入到单体应用预订流的微服务更为方便。航空公司可以开发一个简单推荐服务，并持续新版本迭代，直至它满足需求精度。

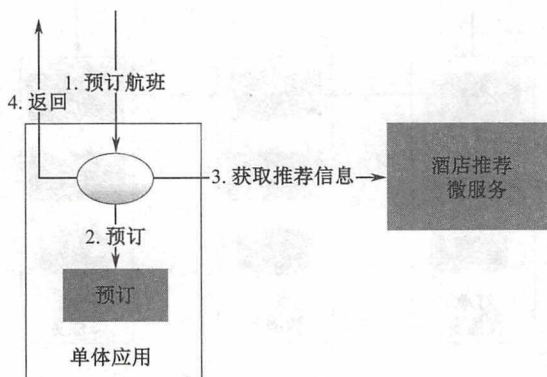


图 1-21

弹性和可扩展性

由于微服务是较小的工作单元，它允许我们实现可扩展性。

在一个应用中，对于不同功能的可扩展性需求可能不同。一个单体应用，包是单个 WAR 或者一个 EAR，只能作为一个整体进行伸缩。当出现高速数据流时，I/O 密集型的功能很容易降低整个应用的服务水平。

微服务中，每一个服务可以独立扩容或缩容。由于扩展性可以选择性地应用于每一个服务，通过微服务方法的扩展代价是相当小的。

实际上，扩展一个应用有很多途径，并且很大程度上受制于应用架构和行为。

Scale Cube 定义了 3 个主要的应用扩展方法：

- (1) 通过水平克隆应用来扩展 x 轴。
- (2) 通过切分不同功能来扩展 y 轴。
- (3) 通过分区或分片数据来扩展 z 轴。

提示

从如下网站获取更多关于 Scale Cube 的信息：<http://theartofscalability.com/>。

对单体应用进行 y 轴扩展后，单体应用根据业务功能被分割成了更小的单元。许多组织成功地应用这项技术来移除单体应用。原则上，其产生的功能单元符合微服务的特性。

例如，一个典型的航空公司网站，统计表明航班搜索与航班预订的比例高达 500:1。这意味着每 500 次搜索事务产生一个预订交易。在这一设想下，搜索需要的可扩展能力比预订功能高 500 倍，这是选择性扩展的理想用例。

解决方案是以不同的方式处理搜索请求和预订请求。对于一个单体应用架构，仅有可能扩展 Scale Cube 中的 z 轴。然而，这样做代价很高，因为所有代码都需要复制。

如图 1-22 所示，搜索和预订定义成不同的微服务从而搜索和预订可以独立。在这个图中，搜索有 3 个实例，而预订有 2 个实例。选择性扩展不限制实例数量。在搜索的例子中，一个内存数据网格(IMDG)，如 Hazelcast，可以用作数据存储。这将进一步提升搜索的性能和可扩展性。当一个新的搜索微服务实例被实例化，额外的

IMDG 节点加入到 IMDG 集群，所有预订微服务实例连接相同的数据库实例。

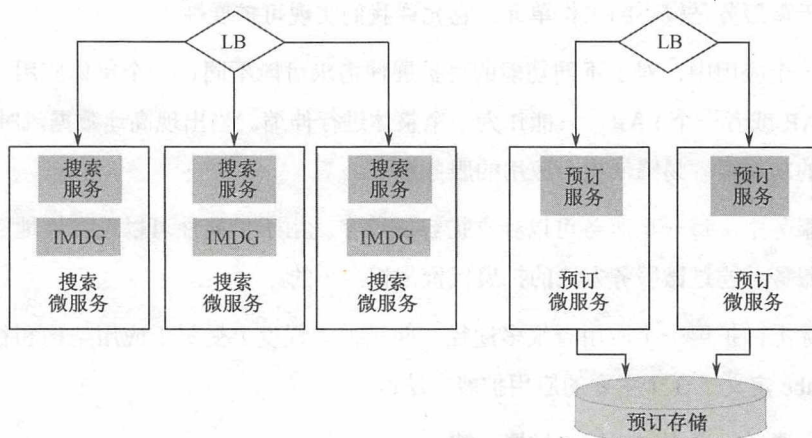


图 1-22

允许替换

微服务是自包含的，独立部署模块允许一个微服务替换另外一个相似的微服务。

许多大型企业遵循自建或外购比较策略去实现软件系统。比较常见的情况是自建软件系统的大部分功能，同时从外部采购一些特定的功能。这个方式在传统单体应用中有很大挑战，因为这些应用组件是高度耦合的。尝试向单体应用中嵌入第三方解决方案会导致复杂集成。对于微服务，这没有后顾之忧。架构上的，一个微服务可以轻易被内部或第三方研发的另一个微服务取代。

在航空业，票务的定价引擎是很复杂的。不同路线的票价使用复杂的称之为定价逻辑的数学公式进行计算。相对于完全自研，航空公司可能选择直接从市场上购买一个定价引擎。在单体架构中，定价是票价和预订的一个功能。大部分情况下，定价、票价和预订是硬关联的，它们几乎不可能分离。

如图 1-23 所示，在精心设计的微服务系统中，预订、票价和定价将是独立的微服务。替换定价微服务对任何其他服务只有最小的影响，因为它们都松散耦合和独立。今天，它可以是第三方服务；明天，它可以很容易被另一个第三方或自建的服务所替代。

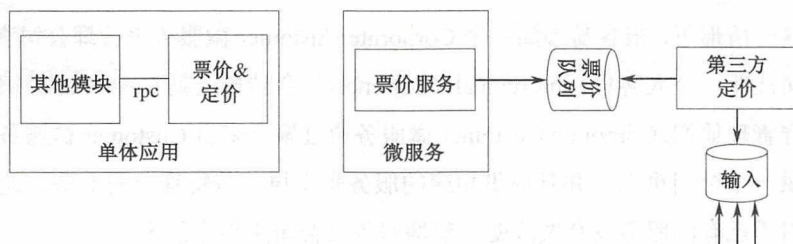


图 1-23

支持构建有机系统

微服务可以帮助我们构建有机系统。当将单体系统逐步迁移成微服务时，这是相当重要的。

有机系统是在一段时间之后通过添加更多功能在上面来横向发展的系统。实际上，一个应用程序在其生命周期会变得难以想象地大，从而导致应用的可管理性显著下降。

微服务都是独立可控的服务。这使我们能够在对现有服务影响最小的情况下不断添加越来越多的服务，构建这样的系统不需要大量的资金投入，对于企业而言是可接受的。

航空公司的一个诚信系统构建于几年前，把个体客户作为目标。直至航空公司开始提供诚信福利给他们的公司客户之前一切都是良好的。客户是在公司下面独立分组的。由于当前系统核心数据模型是扁平的，如果要把个体客户作为目标，公司环境需要改变核心数据模型，因此需要大量的重构来合并这一需求。

如图 1-24 所示，在一个基于微服务的架构中，客户信息会被 Customer 微服务管理，而信用是通过 Loyalty Points 微服务管理。

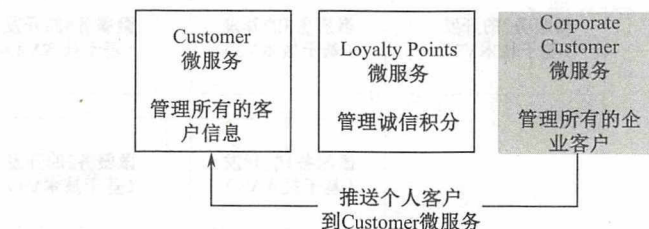


图 1-24

在这一情形下，很容易添加一个 Corporate Customer 微服务来管理公司客户。当一个公司注册，个人客户（Individual Customers）会被推送到 Customer 微服务来像往常一样管理他们。Corporate Customer 微服务通过聚合来自 Customer 微服务的数据对外提供一个公司页面，并且提供相应的服务来支撑一些特殊公司业务。通过这个方法，对于已有的服务没有做改变，新增服务只会带来很小的影响。

帮助减少技术债务

由于微服务在规模上很小，并且只有较小依赖，下线或者替换微服务的代价也是最小的。

技术的改变是软件开发中的一个障碍。在许多传统单体应用中，由于技术的快速改变，如今的应用应该是可以延续使用的。架构师和开发者趋向于添加抽象层的防护来应对技术的改变。然而，实际上这种方法并不能解决问题，反而导致系统的过度设计。由于技术更新经常是冒险的，而且对业务没有直接收益，企业可能不愿意投资来减少应用程序的技术债务。

使用微服务，可以单独更改或升级每个服务的技术，而不是升级整个应用程序。

例如，将写在 EJB 1.1 和 Hibernate 上的 500 万行应用程序升级到 Spring、JPA 和 REST 服务，这几乎类似于重写整个应用程序。在微服务架构中，只需要做增量工作。

如图 1-25 所示，老版本的服务运行于老版本的技术上，新服务开发可以利用最新的技术。相比于优化单体应用，迁移微服务或者下线微服务的代价相当小。

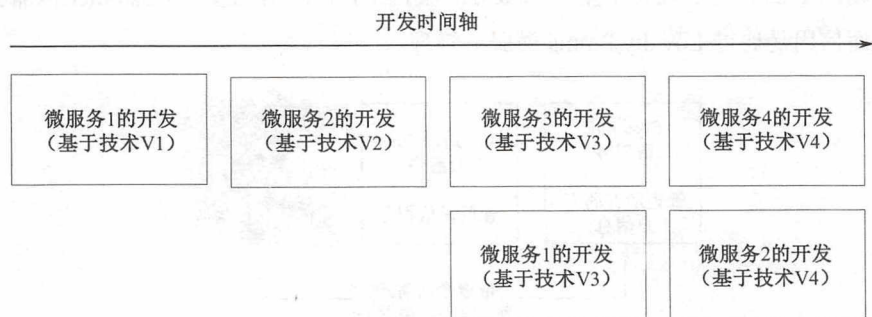


图 1-25

允许不同版本共存

微服务将服务运行时环境与服务本身一起打包，这使得服务的多个版本能够在同一环境中共存。

现实中，会有需要相同服务的不同版本同时运行的情况。零停机升级必须从一个版本平滑地切换到另一个版本。对于单体应用，这是一个复杂的过程，因为更新集群中一个节点的一个新服务的版本是比较烦琐的，而且可能会导致类加载问题（有的时候多版本不能共存的问题）。一个灰度发布，新版本仅开放给少量用户来验证新服务，这是微服务的多版本必须共存的另一个示例。

使用微服务，所有这些情景都容易管理。由于每个微服务使用独立的环境，包括如嵌入的 Tomcat 或 Jetty 等服务监听者，多版本可以发布，而且可以没有任何错误地优雅过渡。当用户查找服务时，需要指定版本。例如，在灰度发布中，一个新的用户接口发布给了用户 A。当用户 A 向微服务发送一个请求，它查找灰度发布版本，其他用户继续查找上一个生产版本。

注意：需要在数据库级别进行仔细检查，以确保数据库设计始终向后兼容，以避免中断更改。

如图 1-26 所示，Customer 服务的版本 1 和版本 2 由于彼此之间互不干涉可以共存，给它们单独地部署环境。可以在网关设置路由规则来转移流量至指定的实例，如下图所示。或者，客户端可以将版本号作为请求本身的一部分。在图中，网关选择的版本基于源请求的所在地区。

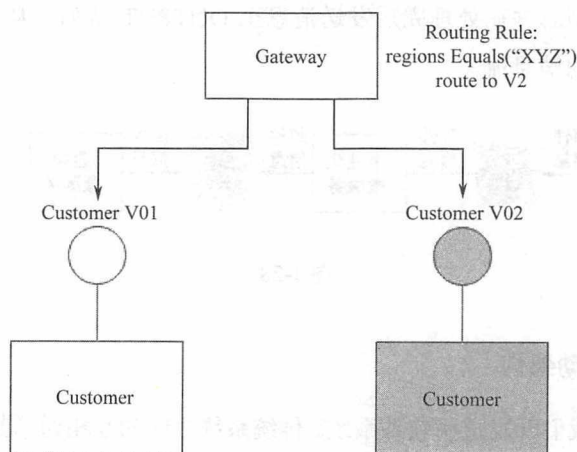


图 1-26

支持自组织系统构建

微服务帮助我们构建自组织系统。一个自组织系统支持良好的自动部署，能复原，拥有自我修复和自我学习能力。

在一个良好架构的微服务系统中，一个服务对于其他服务是未知的。它从指定队列接收消息并处理它。处理结束后，可能发出另外的消息，以触发其他服务。这允许我们删除这个生态系统中的任何微服务，而不用分析对整个系统的影响。不需要额外的代码改变和服务编配，也没有中央大脑去控制和调整这个过程。

想象存在一个通知服务，监听一个 INPUT 队列并发送通知到一个 SMTP 服务器，如图 1-27 所示。

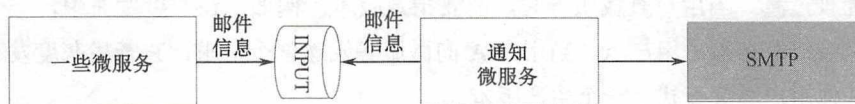


图 1-27

让我们假设，后来需要引入个性化引擎来负责将消息的语言改变为客户的母语，以在将消息发送给客户之前对其进行个性化。

如图 1-28 所示，我们创建一个新的微服务来做这个工作。输入队列会被配置成在外部配置服务器的 INPUT，个性化服务器会从 INPUT 队列（之前，这是用于通知服务）获取消息并在处理完后发送消息至 OUTPUT 队列。从下一刻起，系统自动采用这个新的消息流。

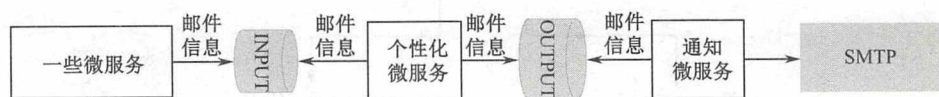


图 1-28

支持事件驱动架构

微服务允许我们开发透明软件系统。传统系统之间的互相通信是通过本地协议，因此运行起来像一个黑盒应用。业务事件和系统事件，是很难理解和分析的。现代

应用需要数据用作业务分析，来理解动态系统行为，分析市场趋势，他们也需要响应实时事件。事件是数据提取的有用机制。

一个架构良好的微服务总是与输入和输出事件一起工作。这些事件可以被任何服务利用。一旦提取了，事件可以使用到多样的案例。

例如，企业想要实时查看按产品类型分类的订单的速度。在一个单机系统，我们需要考虑如何提取这些事件。这可能需要对系统进行变更。

在微服务的世界里，订单事件已在每次订单创建时发布。这意味着只是添加一个新服务来订阅同一主题，提取事件，执行请求的聚合，以及推送另一个事件供仪表板使用，如图 1-29 所示。

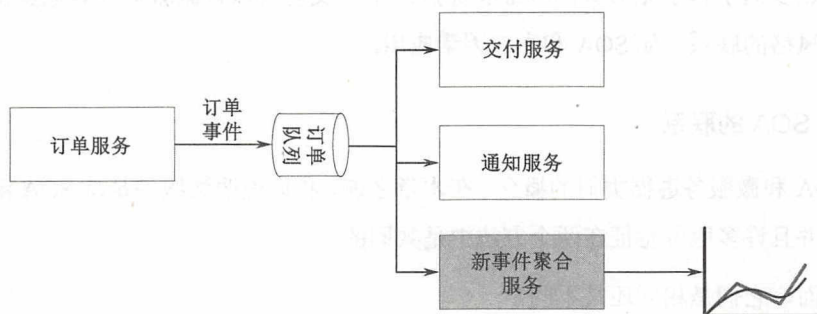


图 1-29

允许 DevOps

微服务是 DevOps 的关键驱动因素之一。DevOps 在许多企业中大量采用和实践，主要是提高交付和敏捷速度。采用 DevOps 需要文化的改变，程序的改变，也需要架构的改变。DevOps 主张敏捷开发，高速发布周期，自动测试，自动提供基础设施和自动部署。

使用传统的单体应用程序实现所有这些过程的自动化是非常困难的。微服务不是最终答案，但是微服务是在众多 DevOps 实现中的舞台中心。许多 DevOps 工具和技术也是围绕微服务的使用来展开的。

考虑到一个单体应用花费数小时完成一个完整编译，再花 20~30min 启动应用，

可以看出这种类型的应用不是理想的 DevOps 自动化,很难在每次提交时做到自动持续集成。由于大型单体应用程序不是自动化友好的,连续测试和部署也是很难实现的。

从另一方面看,小步调的微服务是更加自动化友好的,可以更容易支持这些需求。微服务也允许更小的、专注于敏捷开发的团队,团队会基于微服务的边界来组织。

与其他架构风格的联系

现在我们了解了微服务的特征和优点,本节我们将探索微服务与其他紧密相连的架构风格的联系,如 SOA 和十二因素应用。

与 SOA 的联系

SOA 和微服务遵循类似的概念。在本章之前,我们说明微服务是从 SOA 演变过来的,并且许多服务特征在两个方法中是共同的。

然而,它们是相同还是不同?

由于微服务是从 SOA 演进过来的,微服务的许多特征类似于 SOA。让我们首先考究下 SOA 的定义。

来自开发组织联盟对 SOA 的定义如下:

面向服务架构(SOA)是一种支持服务导向的架构样式。服务导向是一种从服务、基于服务的开发及服务产出的角度来思考分析问题的方式:有指定结果的重复业务活动的一个逻辑呈现(如检查客户信用,提供天气数据,巩固训练报告),它是自包含的,可能由其他服务组成。对于服务消费者它是一个“黑盒”。

面向服务集成

面向服务集成是指给许多组织使用的基于服务的集成方法。

许多组织可能已经用过 SOA 来解决他们的集成复杂性。通常,这被称为面向服务的集成(SOI)。如图 1-30 所示,在这个例子中,应用间通过一个标准协议和消息

格式的公用集成层互相通信,如通过 HTTP 或 JMS 的基于 SOAP/XML 的 Web 服务。这些类型的组织关注于企业集成模式(EIP)模式化他们的集成需求。这种方法强烈依赖于重量级 ESB,如 TIBCO Business Works、WebSphere ESB、Oracle ESB 及类似产品。大多数 ESB 供应商也打包一系列相关产品,如规则引擎、业务过程管理引擎等,来作为一个 SOA 套件。这些组织的集成深度根植于他们的产品中,他们在 ESB 层编写繁重编排逻辑,或者在服务总线中编写业务逻辑本身,并且通过 ESB 部署和访问所有企业服务,然后通过一个企业治理模块来管理。对于这种组织,微服务是与 SOA 有本质的不同。

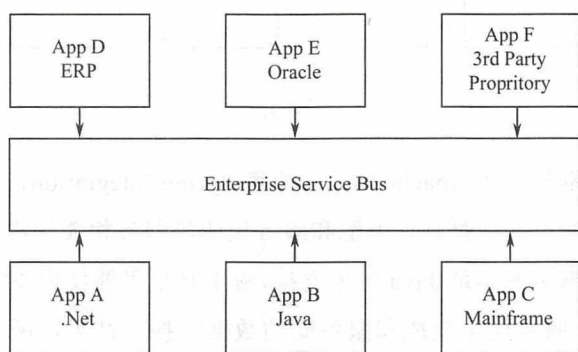


图 1-30

旧资产现代化

SOA 也用于在旧资产应用之上构建服务层。

如图 1-31 所示,另一种类型的组织可能使用 SOA 来转化项目或旧资产现代化项目。构建服务并部署在 ESB 层,用 ESB 适配器去连接后端系统。对于这些组织,微服务不同于 SOA。

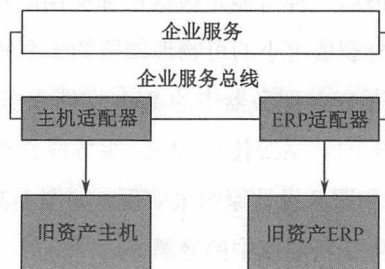


图 1-31

面向服务应用

一些企业在应用层采用 SOA，如图 1-32 所示。

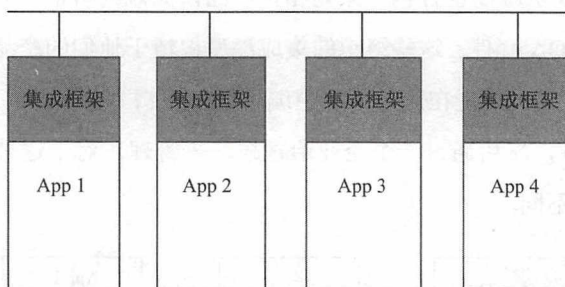


图 1-32

轻量级集成框架，如 Apache Camel 或者 Spring Integration，嵌入在应用中去处理诸如协议适配、并行执行、编配和服务集成等服务相关的跨切面功能。由于一些轻量级集成框架有本地 Java 对象支持，应用甚至可能使用本地 Plain Old Java Objects (POJO) 服务用于集成和服务间的数据交换。结果，所有服务不得不被打包成一个 Web 存档文件。这样的组织可能把微服务作为他们 SOA 的下一个发展阶段。

使用 SOA 单体迁移

最后一种可能是根据单体应用的转换点将单体应用转换成更小的单元。他们将应用分裂成更小的可物理部署的子系统，类似于之前解释的 y 轴扩展方法，并将它们作为 Web 服务器中的 Web 文档或作为本地生产的容器的 JARs 进行部署。这些作为服务的子系统使用 Web 服务或其他轻量级协议来交换服务间数据，也可以使用 SOA 和服务设计原则来实现，如图 1-33 所示。对于这种组织，他们可能趋向于认为微服务是新酒瓶中的老酒。

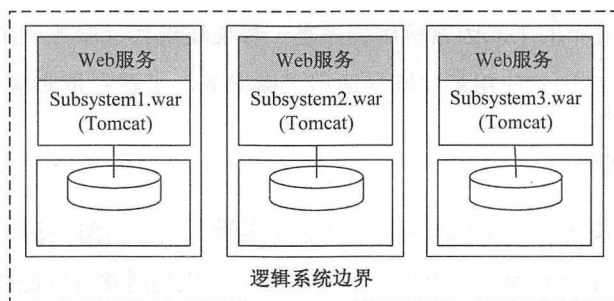


图 1-33

与十二因素的联系

云计算是快速发展的技术之一。云计算带来许多益处，如成本优势、速度、敏捷性、灵活性和弹性。云服务可以降低企业的运营成本，吸引了大批企业客户，不同的云供应商，如 AWS、Microsoft、Rackspace、IBM、Google 等，使用不同的工具、技术和服务。另外，企业意识到这种不断演变的战场，因此他们正在寻找选项来降低依赖单一供应商的风险。

许多组织将自己的服务部署到云上面。在这种情况下，应用可能并没有认识到云平台提供的所有益处。在移到云上之前，一些应用需要经历彻底检修，另外一些可能只需要轻微调整。这取决于应用程序是如何架构和开发的。

例如，如果应用程序将其生产数据库服务器 URL 硬编码为应用程序 WAR 的一部分，则在将应用程序移动到云之前需要对其进行修改。在云中，基础架构对应用程序是透明的，特别是不能假定物理 IP 地址。

我们如何确保应用程序甚至微服务可以跨多个云提供商无缝运行，并利用云服务的优势（如弹性）？

在开发云本地应用程序时遵循某些原则是很重要的。

提示

原生云是应用开发的一个术语。这类应用可以在云环境中有效地工作，理解和利用诸如弹性、按量计费、失败感知等云特征。

由 Heroku 发布的十二因素应用程序是一种描述现代云服务应用程序所期望的特性的方法，十二因素应用程序同样适用于微服务，了解它是非常重要的。

一份基准代码

基准代码原则建议每个应用有一个单独的基准代码，如图 1-34 所示。相同基准代码可以部署在多个环境，如开发、测试和生产。代码通常有资源控制系统管理，如 Git、Subversion 等。

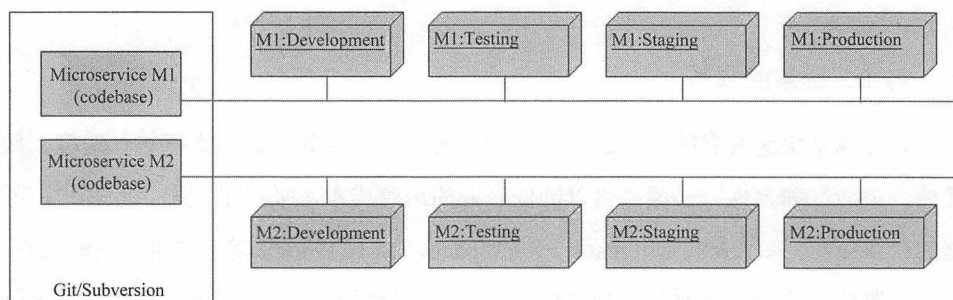


图 1-34

同理，每个微服务应该有自己的基准代码，这个基准代码不与任何其他微服务共享。

捆绑依赖

根据这个原则，所有应用应该捆绑其依赖与应用程序包。如图 1-35 所示，使用诸如 Maven 和 Gradle 构建工具，我们在 pom.xml 或 gradle 文件中显式管理依赖关系，并使用诸如 Nexus 或 Archiva 中央构建库将其链接起来，从而保证版本被正确管理。最终可执行文件会被打包成一个 WAR 包或 JAR 包，且内嵌了所有依赖。

每个微服务应该在最终可执行程序包中捆绑所有需要的依赖，以及诸如 HTTP 监听器等执行库。在微服务环境中，这是需要遵循的基本原则之一。

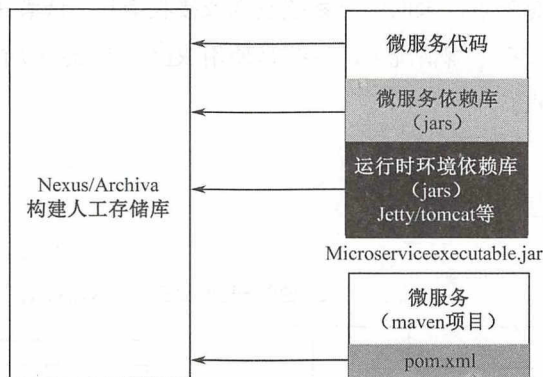


图 1-35

外部配置

这一原则建议将代码中所有配置参数外部化。应用程序的配置参数因环境而异，如支持电子邮件 ID 或外部系统的 URL、用户名、密码、队列名称等，在开发、测试和生产时会有不同。

同样的原则对于微服务也是适用的。如图 1-36 所示，微服务的配置参数应该从外部资源中加载。自动发布和部署程序时，将配置参数作为不同环境间的唯一不同点。

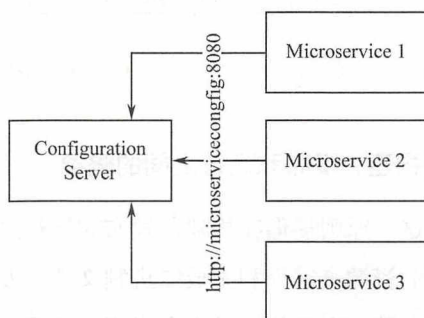


图 1-36

可寻址的后端服务

所有的微服务都应该可以通过一个可寻址的 URL 来访问。所有服务在它们的执行生命周期内需要与一些外部资源进行交流。例如，它们可能会监听或发送消息到一个消息系统、发送电子邮件、持久化数据到数据库等。所有这些服务应该可以通过一个 URL 获取，而不需要复杂的通信需求。

如图 1-37 所示，在微服务的世界里，微服务通过 REST 和 JSON 的 HTTP 协议

或基于 HTTP 的消息端点，来向消息系统发送或接收消息，或者用其他服务的 APIs 来接收或发送消息。在常规情况下，这些是使用 REST 和 JSON 的 HTTP 端点或基于 HTTP 的消息端点。

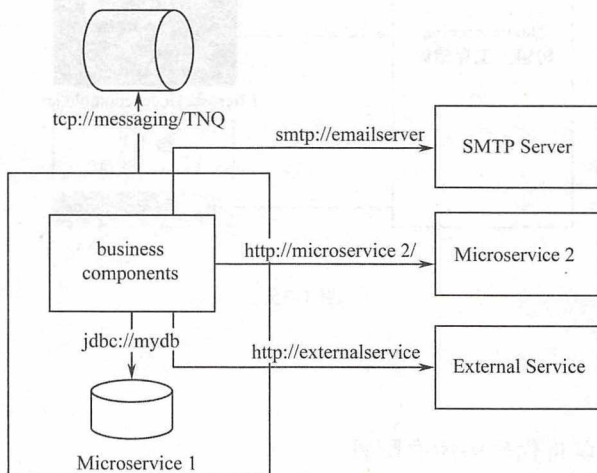


图 1-37

构建、发布和运行之间的隔离

这一原则提倡在构建、发布和运行阶段的强隔离。构建阶段是指通过包含所有需要的资源来编译和生成二进制文件。发布阶段是指整合二进制文件与指定环境的配置参数。运行阶段是指在指定执行环境上运行应用。这一过程是单向的，所以不可能从运行阶段向后传播变更信息到构建阶段。但也不建议专门为生产做特殊构建，应该遵循管道的流程。

如图 1-38 所示，在微服务中，构建阶段会创建可执行 JAR 包，包括诸如 HTTP 监听器的服务运行时环境。在发布阶段，这些可执行文件会结合诸如生产 URLs 等发布配置并创建发布版本，大多数时候是一个类似于 Docker 的容器。在运行阶段，这些容器将会通过容器调度器部署在生产上。

无状态，无共享进程

这个原则建议进程应该是无状态且无共享内容的。无状态的应用可以容错并可

以轻松扩容。

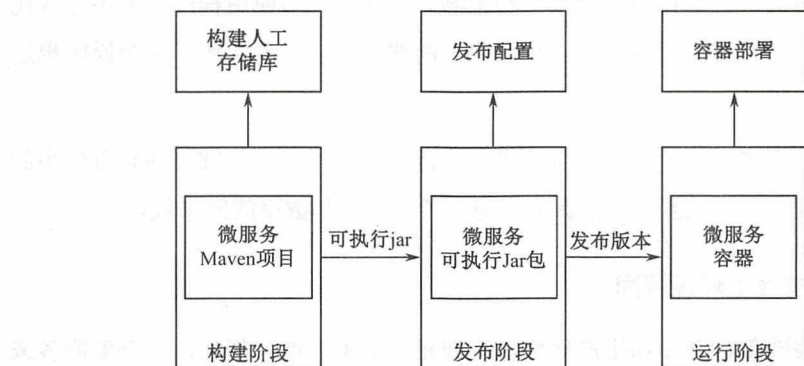


图 1-38

所有的微服务应该设计无状态功能，如果有任何一个需求存储了状态，它应该通过后端数据库或内存缓存来完成。

通过端口绑定暴露服务

一个十二因素应用是期望成为自包含的。在传统意义上，应用是被部署在一个 Web 服务器或应用服务器上的，如 Apache Tomcat 或 Jboss。十二因素应用并不依赖于一个外部 Web 服务器，如 Tomcat 或 Jetty 等 HTTP 监听器已经被嵌入到服务本身。

端口绑定是微服务自动化和自包含的基本要求之一。

并发扩容

这个原则声明程序应该被设计成可以通过复制的方式进行扩容，即使用多个进程。

在微服务世界，服务设计成向外扩展而不是向上扩展。 x 轴扩容技术主要用于通过启动部署多个相同的服务实例来扩展服务。服务可以基于传输流量进行弹性扩容或缩容。此外，微服务可以利用并行处理和并发框架来进一步加快事务处理。

易处理

这一原则提倡构建具有最小启动和停止时间并支持优雅停机的应用。在一个自

动部署环境中，我们应该尽可能快地启动和关停一个实例。如果应用的启动和关停花费相当多的时间，会给自动化带来不利影响。启动时间与应用程序的大小是成比例的。在面向自动扩展的云环境中，我们应该能够快速启动新实例。这个原则也适用于升级新服务版本。

在微服务环境中，为了实现全自动化，保持应用规模尽可能细小并具有最小的启动和关停时间是相当重要的。微服务也应该考虑对象和数据的懒加载。

开发环境与线上环境等同

这一原则表明保持开发和生产环境尽可能相同的重要性。例如，一个多服务或多进程的服务，如任务调度服务、缓存服务。在开发环境，我们常常在一个机器上运行所有服务，然而在生产环境，我们会将每个服务部署在独立的机器上。这主要是为了管理基础设施的成本。缺点是如果生产上发生故障，没有完全相同的环境来重现并修复问题。

这一原则不仅适用于微服务，也适用于任何应用开发。

外部化日志

十二因素应用从不尝试存储或发送日志文件。在云中，最好避免本地 I/O。如果 I/O 在给定的基础设施中不够快，它可能导致瓶颈。这个问题的解决方案是使用集中日志框架进行日志搜集和分析，例如 Splunk、Greylog、Logstash、Logplex 和 Loggly

将日志发送到中央存储库，单击 logback 追加器并写入其中一个日志框架所在端口。

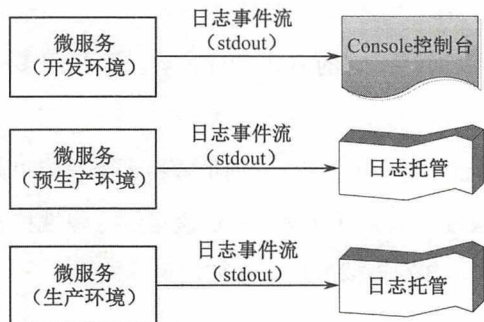


图 1-39

如图 1-39 所示，在一个微服务生态系统中，我们将一个系统拆分成若干小服务是非常重要的，这可能导致日志是分散的。如果它们存储日志到本地，关联不同服务间的日志会变得极其困难。

在开发中，微服务可能将日志流指向 `stdout`，然而在生产中，这些流会被日志托管工具捕获并发送到中央日志服务进行存储和分析。

打包管理流程

除了应用程序服务，大多数应用程序还提供管理任务。这个原则建议对应用程序服务和管理工作使用相同的发布包及部署在相同的环境。也就是说，管理代码应该与应用代码一起打包。

这个原则不仅对微服务有效，而且它也适用于任何应用程序开发。

微服务使用案例

微服务不是灵丹妙药，不会解决当今世界的所有架构挑战。使用微服务时没有硬性规则或严格的指导方针。

微服务可能不会适合每个案例，微服务的成功很大程度上依赖于案例的选择。使用微服务之前，最好先在案例中去一一核验微服务的好处，前面章节讨论过的所有微服务的好处都必须覆盖到。如果没有可量化的好处，或者成本高于好处，那么这个案例可能不是使用微服务的最佳选择。

让我们讨论一些通常使用的适用于选择微服务架构的场景：

- 迁移单体应用，由于要提高在可扩展性、可管理性、敏捷性或交付速度上的需求。另一个类似的场景是重写开发生命周期已经结束，但是仍在被大量使用的应用。在这两个用例中，微服务提供了一个机会。使用微服务架构，可以通过慢慢地将功能转换成微服务来重构一个遗留应用。这种方法大有好处。不需要大量预支投入，对业务没有重大影响，也没有严重的商业风险。由于服务依赖是已知的，微服务依赖可以被很好地管理。
- 实用计算场景，如集成优化服务、预估服务、计价服务、预测服务、提议服务、推荐服务等都是微服务的很好选择。这些是独立无状态的计算单元，它接收确切数据和应用算法后返回相应结果。独立的技术服务，如通信服

务、加密服务、认证服务等也适用于采取微服务。

- 在许多情况下，我们可以构建自动化的无图形界面的业务应用或服务，如支付服务、登录服务、航班搜索服务、客户资料服务和通知服务等。这些服务在多个应用场景中重复使用，因此用微服务来构建它们是很好的选择。
- 可能有微型或大型应用程序负责单一功能，如简单的事件追踪应用。它所需要做的是获取时间、持续时间，并执行任务。通常的企业应用也可以考虑使用微服务。
- 架构良好的 MVC Web 应用 [后端即服务 (BaaS) 场景] 的后端服务按需加载数据。也就是说，数据可能来自多个逻辑上不同的数据源，就像之前提到的 *Fly By Points* 例子。
- 高度敏捷的应用程序，要求交付速度或上线时间的应用程序，创新试点的 DevOps 应用程序，创新系统类型的应用程序等也可以选择微服务架构。
- 我们可以预知的、会从微服务获益的应用程序，如多语言需求及需要命令和查询责任分离 (CQRS) 的应用程序等，也是微服务架构的潜在候选者。

有少量场景我们应该避免使用微服务：

- 如果企业的策略被迫使用集中管理的重量级组件（如 ESB）来托管业务逻辑，或是有违反微服务基本原则的策略，那么微服务不是正确的解决方案，除非放宽。
- 如果企业的文化、流程等基于传统的瀑布开发模式，冗长的发布周期，矩阵团队，手动部署和烦琐的发布流程，没有基础设施配置等，那么微服务可能不适合。这是由康威定律支持的。这表明企业架构和软件息息相关。

提示

有关康威定律的更多信息，请访问：http://www.melconway.com/Home/Conways_Law.html。

微服务早期采用者

许多企业已经成功地踏上了他们的微服务世界之旅。本节我们将研究微服务领

域的一些前沿者，分析他们为什么这样做及他们做了什么。我们将在最后进行一些分析并得出结论。

- Netflix (www.netflix.com): 一个国际点播媒体流公司，是微服务领域的先驱。Netflix 将原先开发传统单体应用代码的开发者拆分成更小的微服务开发团队。这些微服务一起为数百万 Netflix 用户提供数字流媒体服务。在 Netflix，工程师开始用单体应用，吸取经验和教训之后根据业务功能，将应用分割成了松耦合的更小单元。
- Uber (www.uber.com): 一个国际运输网络公司，2008 年开始使用具有单个基准代码的单体架构。所有服务嵌入到单体应用程序。当 Uber 的业务扩展到多个城市，挑战开始了。Uber 之后基于 SOA 架构将系统拆分成更小独立单元。每个模块交给不同的团队并赋予他们选择语言、架构和数据库的权利。Uber 有许多微服务使用 RPC 和 REST 部署在他们的生态系统。
- Airbnb (www.airbnb.com): 一个提供值得信赖的住宿市场的世界领导者，开始是用一个单体应用执行业务需要的所有功能。随着流量的增加，Airbnb 面临着扩展的问题。单个基准代码变得太复杂，无法管理，导致关注分离不佳，并遇到性能问题。Airbnb 将他们的整体应用程序打破成更小的部分，单独的基准代码在单独的机器上运行，具有单独的部署周期。Airbnb 在这些服务上开发了自己的微服务和 SOA 生态系统。
- Orbitz (www.orbitz.com): 一个在线旅行门户，从 2000 年的一个单体应用架构开始，它有一个 Web 层、一个业务层和一个数据库层。由于业务的扩展，他们面临着分层单体架构的可管理性和扩展性问题。Orbitz 经历了连续的架构挑战，之后，Orbitz 将他们的单体应用拆分成许多更小的应用。
- eBay (www.ebay.com): 最大的在线零售商之一，开始于 20 世纪 90 年代后期，使用的是 Perl 应用和 FreeBSD 平台。eBay 随着业务增长遇到了扩容问题。他一直在改善架构上投资。在 2000 年代中期，eBay 将业务拆分成基于 Java 和 Web 服务的更小的系统。他们使用数据库分区和功能隔离来满足服务的可扩展性。
- Amazon (www.amazon.com): 最大的在线零售网站之一，在 2001 年是运行在一个 C++ 写的大型单体应用上。良好架构的单体应用是基于许多模块

组件的分层架构。然而所有这些组件是高度耦合的。结果，Amazon 并不能通过拆分团队至更小的组来加速他们的开发周期。Amazon 之后将代码分离作为独立的功能服务，用 Web 服务封装，最终发展成微服务。

- Gilt (www.gilt.com): 一个在线购物网站，始于 2007 年一个分层单体 Rails 应用，在后端使用 Postgres 数据库。类似于其他大多数应用，随着流量的增加，Web 应用程序无法提供所需的恢复能力。Gilt 通过引入 Java 和多语言持久化经历了一次架构大修改。之后，Gilt 使用微服务的概念将原有应用重构成许多小应用。
- Twitter (www.twitter.com): 最大的社交网站之一，在 2000 年代中期，开始用一个三层单体 rails 应用。之后，当 Twitter 经历了用户群体的增长，他们经历了一个架构重构周期。这次重构，Twitter 从一个传统 Web 应用迁移成基于 API 事件驱动的核心。Twitter 使用 Scala 和 Java 来开发多语言持久化的微服务。
- Nike (www.nike.com): 服装和球类的世界领导者，类似于许多其他公司，Nike 运行的也是已经非常稳定的旧应用。后来，Nike 转向重量级的商业产品，目的是稳定传统应用程序，但是这些应用程序规模庞大，发布周期长，需要太多的手动工作来部署和管理应用程序。之后，Nike 转到了基于微服务的架构，并降低了相当多的开发周期。

共同的主题是整体迁移

我们分析前面的企业时，有一个共同的主题，这些企业都是开始使用单体应用程序，然后体会到了先前版本的痛点并从中学习，最终将应用转换到了微服务架构。

即使在今天，许多初创公司还是从单体应用开始，因为它很容易启动、也很容易概念化，然后在需求出现时逐渐迁移到微服务。单体到微服务迁移场景有一个额外的优势：它们具有所有的信息，可以重构。

尽管对于这些企业而言都是整体转型，但是不同企业的转型催化剂大相径庭。一些常见的转型动机是缺乏可扩展性、长的开发周期、过程自动化、可管理性和业务模型的变化。

整体迁移是不聪明的，其实是有机会从一开始就构建微服务，更好过于构建一个大系统，并寻求机会去构建可以快速赢得业务的更小服务。例如，向航空公司的端到端货物管理系统添加卡车服务，或者向零售商的忠诚度添加客户评分服务系统，这些可以用微服务来实现各自的功能。

另外，许多企业只是将微服务用于其关键业务的客户端应用程序，原来的单体应用依旧保留。

另一个重要的现象是大多数组织在他们的微服务进程中处于不同的成熟水平。当 eBay 在 21 世纪初从一个单体应用程序转换时，他们将应用程序功能分为更小、独立和可部署的单元。这些逻辑划分的单元被封装成 Web 服务，虽然单一责任和自主是他们的基本原则，但是架构仅限于当时开发的技术和工具，如 WebService。Netflix 和 Airbnb 等企业也是用他们自己拥有的构建方法去解决他们所面临的具体挑战。总而言之，所有这些都不是真正的微服务，而是规模小、业务保持一致的服务，遵循相同的特征。

“明确或终极微服务”是无状态的，这是通过一天天的演变而成熟起来的。架构师和开发者的口头禅是可替换原则：构建一个最大化更换其部件的能力并最大限度地降低更换其部件的成本的架构。底线是企业不应该不清楚自己的需求，随波逐流开发微服务。

总结

本章我们学习了微服务基础知识的几个例子。

我们探索了微服务从传统的单体应用程序的演变，并且研究了现代应用程序架构所需的一些原则和心态转移。我们还说明了微服务和用例的特点和好处。在本章中，我们探索了微服务与面向服务架构和十二因素应用程序的关系。最后，我们分析了来自不同行业的几个企业的例子。

我们将在第 2 章中开发几个示例微服务，以便对理论有更深入的理解。

第 2 章

用 Spring Boot 构建微服务



由于 Spring Boot 框架的强大，开发微服务不再冗长乏味。Spring Boot 是用 Java 来开发可用于生产环境的微服务的框架。

本章将通过实际代码，将上一章中学习的关于微服务的理论知识学以致用。本章将介绍 Spring Boot 框架，并解释 Spring Boot 如何帮助构建符合前一章讨论的原则和特性的 RESTful 微服务。最后，Spring Boot 提供的一些功能将使微服务用于实际生产中。

本章结束时，您会了解到：

- 设置最新的 Spring 开发环境。
- 使用 Spring 框架开发 RESTful 服务。
- 使用 Spring Boot 构建完全合格的微服务。
- 使用 Spring Boot 特性来构建符合生产需求的微服务。

设置开发环境

为了具体化微服务的概念，将会构建几个微服务。需要提前安装：

- JDK1.8： <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>。
- Spring Tool Suite 3.7.2: <https://spring.io/tools/sts/all>。
- Maven 3.3.1: <https://maven.apache.org/download.cgi>。

也使用其他 IDE，如 IntelliJ IDEA，NetBeans 或 Eclipse。类似地，可以使用诸如 Gradle 的替代构建工具。本章中使用 Maven 库，需要保证类路径和其他路径变量设置正确。

本章基于如下 Spring 库版本：

- Spring Framework 4.2.6.RELEASE。
- Spring Boot 1.3.5.RELEASE。

提示

在本书的前言中提到了下载代码包的详细步骤。

书中的代码也托管在 GitHub 上，网址为 <https://github.com/PacktPublishing/Spring-Microservices>。

作者团队还有其他书籍和视频，里面的代码也托管在 GitHub 上，网址为 <https://github.com/PacktPublishing>。

开发 RESTful 服务——传统方法

我们首先回顾传统的 RESTful 服务开发，然后深入到 Spring Boot。

开发 REST/JSON 服务的过程中使用了 STS。

提示

此示例的完整源代码可在代码文件中的 legacyrest 项目中获取。

下面是开发第一个 RESTful 服务的步骤：

- (1) 启动 STS 并为此项目设置一个工作区。
- (2) 单击菜单项 File|New|Project。
- (3) 选择如图 2-1 (a) 所示的 Spring Legacy Project，单击 Next。
- (4) 选择如图 2-1 (b) 所示对话框中的 Spring MVC Project，单击 Next。

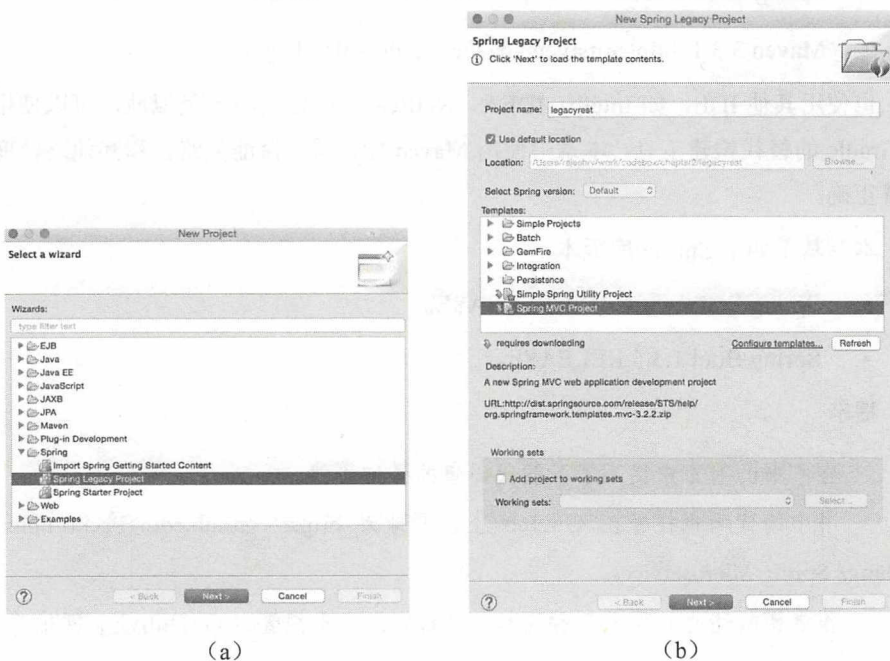


图 2-1

- (5) 选择顶级包名称，如 org.rvslab.chapter2.legacyrest。
- (6) 然后，单击 Finish。
- (7) 此时在 STS 工作区中创建了一个名为 legacyrest 的项目。下一步，编辑 pom.xml。
- (8) 将 Spring 版本号修改为 4.2.6.RELEASE，如下：

```
<org.springframework-version>
```

4.2.6.RELEASE

```
</org.springframework>
```

(9) 在 pom.xml 文件中添加 Jackson 依赖, 用于 JSON-to-POJO 和 POJO-to-JSON 转换。注意: 使用 2.*.* 的版本适用于 Spring4。

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.6.4</version>
</dependency>
```

(10) 在 Java Resources, legacyrest 下面, 展开包并打开默认 HomeController.java 文件, 如图 2-2 所示。

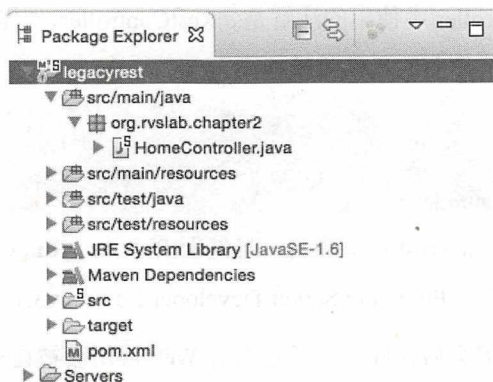


图 2-2

(11) 这里, 我们采用 MVC 架构。重写 HomeController.java 来返回一个 JSON 值, 用于成功响应 REST 请求。

```
@RestController
public class HomeController {
    @RequestMapping("/")
    public Greet sayHello() {
        return new Greet("Hello World!");
    }
}

class Greet {
```



```
private String message;
public Greet(String message) {
    this.message = message;
}
// add getter and setter
}
```

代码里有两个类：

- Greet: 表示一个数据对象的简单的 Java 类，带有 getters 和 setters。类中仅有一个 message 属性。
- HomeController.java: 仅有一个用来处理 HTTP 请求的 Spring 控制器 REST 端点。

注意：HomeController 中使用的注解是 @RestController，它自动注入 @Controller 和 @ResponseBody，与下面的代码效果相同。

```
@Controller
@ResponseBody
public class HomeController {}
```

(12) 右键单击 legacyrest 启动项目，导航至 Run As | Run On Server，然后选择 STS 自带的默认服务器（Pivotal tc Server Developer Edition v3.1）。

正常情况下服务器会启动成功，并且当前 Web 应用会被自动部署到这个 TC 服务器上。

如果服务器启动正常，控制台上会显示如下消息：

```
INFO: org.springframework.web.servlet.DispatcherServlet - FrameworkServlet 'appServlet':
initialization completed in 906ms May 08, 2016 8:22:48 PM org.apache.catalina.startup.Catalina
start
INFO: Server startup in 2289 ms
```

(13) 如果启动成功，STS 会打开链接 <http://localhost:8080/legacyrest/> 并显示 JSON 对象。右键单击 legacyrest | Properties | Web Project Settings 并检查 Context Root 同 Web 应用的上下文根路径是否一致，如图 2-3 所示。

也可以使用 Maven 构建。右键单击项目并单击菜单项 Run As | Maven install。目标文件夹下会生成 chapter2-1.0.0-BUILDSNAPSHOT.war。这个 war 可以部署在任何

servlet 容器中，如 Tomcat、Jboss 等。

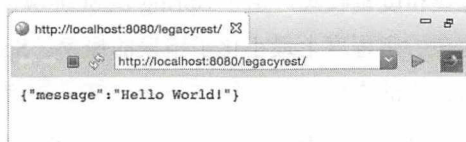


图 2-3

传统 Web 应用转移到微服务

仔细检查前面的 RESTful 服务将揭示这是否真的符合微服务的条件。乍一看，上述 RESTful 服务是一个完全合格的可交互的 REST / JSON 服务。然而，它本质上不是完全自主的。这主要是因为服务依赖于底层应用程序服务器或 Web 容器。在前面的示例中，我们在 Tomcat 服务器上显式创建并部署了一个 war。

这是一种将 RESTful 服务作为 Web 应用程序开发的传统方法。然而，从微服务的角度来看，需要一种机制来将服务开发成一个可执行文件，换句话说就是具有嵌入式 HTTP 侦听器的自包含 JAR 文件。

Spring Boot 是一种允许轻松开发此类服务的工具。类似的还有 Dropwizard 和 WildFly Swarm。

使用 Spring Boot 构建 RESTful 微服务

Spring Boot 是 Spring 团队的一个实用框架，可以快速轻松地引导开发基于 Spring 的应用程序和微服务。该框架使用一种特定的方式来进行配置，从而减少编写大量样板代码和配置所需的工作量。基于 80-20 原则，开发人员应该能够使用许多默认值启动各种 Spring 应用程序。Spring Boot 进一步为开发人员提供了通过覆盖自动配置的值来定制应用程序的机会。

Spring Boot 不仅提高了开发速度，而且还内嵌了一组可直接植入生产环境的运营功能，如运行状况检查和度量收集。由于 Spring Boot 屏蔽了许多配置参数并抽象了许多低级实现，因此在一定程度上最小化了错误的概率。Spring Boot 基于类路径中可用的库来识别应用程序的本质，并运行这些库中打包的自动配置类。

通常，许多开发人员错误地认为 Spring Boot 是一个代码生成器，但事实并非如此。Spring Boot 只能自动配置构建文件，如在 Maven 的情况下为 POM 文件。它还基于数据源属性等默认值设置启动属性，代码如下：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <scope>runtime</scope>
</dependency>
```

例如，在前面的例子中，Spring Boot 知道在项目中使用 Spring Data JPA 和 HSQL 数据库。它自动配置驱动程序类和其他连接参数。

Spring Boot 的一个伟大成果是它几乎消除了对传统 XML 配置的依赖。Spring Boot 通过将所有运行时必需的依赖包装在一个可执行的 JAR 文件中来实现微服务的开发。

开始使用 Spring Boot

有不同方法可以启动基于 Spring Boot 开发的应用程序：

- 使用 Spring Boot CLI 命令行工具。
- 使用支持 Spring Boot 的 IDE，如 STS。
- 在 <http://start.spring.io> 上使用 Spring 初始化项目。

本章中会基于上面的 3 种启动方法，开发各种样本服务。

使用 CLI 开发 Spring Boot 微服务

开发和演示 Spring Boots 功能最简单的方法是使用命令行工具 Spring Boot CLI。按如下步骤执行：

(1) 从 <http://repo.spring.io/release/org/springframework/boot/spring-boot-cli/1.3.5.RELEASE/spring-boot-cli-1.3.5.RELEASE-bin.zip> 下载 spring-boot-cli-1.3.5.RELEASE-bin.zip 文件来安装 Spring Boot 命令行。

(2) 解压文件到指定目录。打开一个命令行窗口并切换到 bin 目录。确保 bin 目录加入到系统路径，这样 Spring Boot 可以从任何位置运行。

(3) 使用以下命令验证安装。如果成功，控制台会打印 Spring CLI 版本号：

```
$spring -version
Spring CLI v1.3.5.RELEASE
```

(4) 下一步，在 Groovy 中快速开发一个 REST 服务，它在 Spring Boot 中是开箱即用的。将下列代码保存在任何目录下的 myfirstapp.groovy 文件：

```
@RestController
class HelloWorldController {
    @RequestMapping("/")
    String sayHello() {
        "Hello World!"
    }
}
```

(5) 为了运行这个 Groovy 应用，进入到保存 myfirstapp.groovy 文件的目录并执行如下命令。服务启动日志的最后几行会类似如下：

```
$spring run myfirstapp.groovy
2016-05-09 18:13:55.351 INFO 35861 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : FrameworkServlet'dispatcherServlet': initialization started
2016-05-09 18:13:55.375 INFO 35861 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : FrameworkServlet'116'dispatcherServlet': initialization completed in 24 ms
```

(6) 打开浏览器窗口并进入到 <http://localhost:8080>；浏览器会显示如下消息：

Hello World!

这个过程中没有创建 war 文件，没有运行 Tomcat 服务器。Spring Boot 自动将 Tomcat 作为 Web 服务器并将其嵌入到应用程序中。这是非常基础的、小型的微服务。之前代码中使用的 `@RestController` 注解，会在接下来的例子中详细研究。

使用 STS 开发 Spring Boot Java 微服务

本节将演示使用 STS 开发另一种基于 Java 的 REST / JSON Spring Boot 服务。

提示

本例的完整源代码可以在本书的代码文件 `chapter2.bootrest` 中获取。

(1) 打开 STS，右键单击 Project Explorer 窗口，导航至 `New|Project`，然后选择 `Spring Starter Project`，如图 2-4 所示，再单击 `Next`。

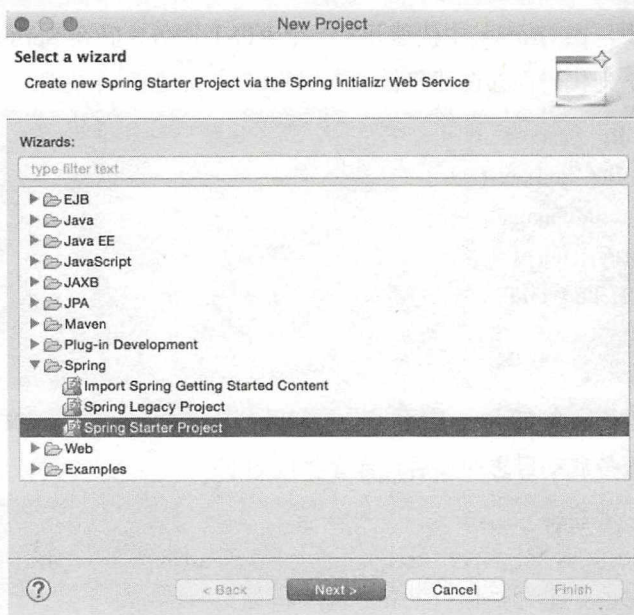


图 2-4

Spring Starter Project 是一个基础的模板向导，提供了许多其他启动库可供选择。

(2) 将项目名称输入为 `chapter2.bootrest` 或者您选择的任何其他名称。Packaging 处选择打包成 JAR 是非常重要的。在传统 Web 应用中，会创建一个 war 文件然后部署到一个 servlet 容器，而 Spring Boot 将所有依赖打包到一个嵌入有 HTTP 监听器的自包含的自主 JAR 文件中。

(3) 选择 Java Version 为 1.8。Spring4 应用建议 Java 1.8。改变其他 Maven 属性，如 Group、Artifact 和 Package，如图 2-5 所示。

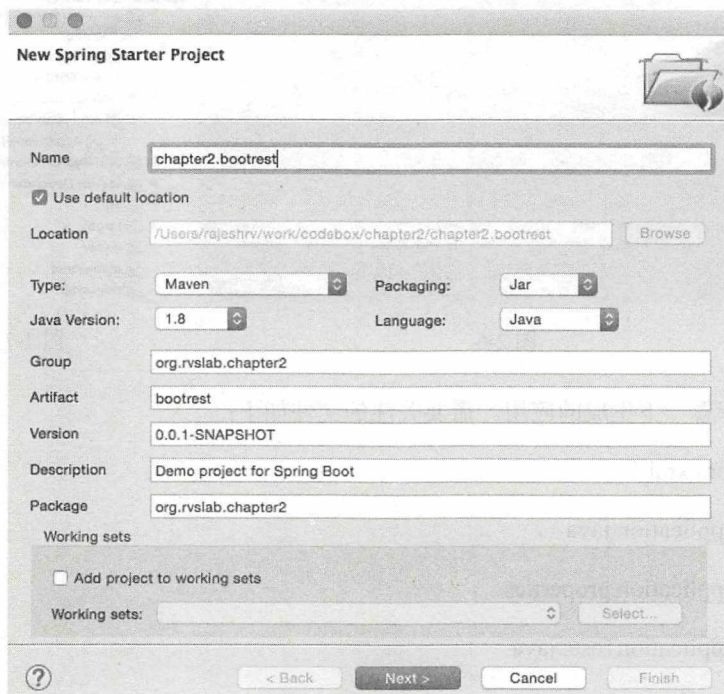


图 2-5

(4) 单击 Next。

(5) 向导程序会显示库选项。在这个例子中，由于是开发 REST 服务。这是告诉 Spring Boot 正在开发 Spring MVC Web 应用程序，以便 Spring Boot 可以根据需要包括必要的库，包括 Tomcat 作为 HTTP 侦听器和其他配置。

(6) 单击 Finish，会在 STS 的 Project Explorer 中生成一个名为 `chapter2.bootrest` 的项目，如图 2-6 和图 2-7 所示。

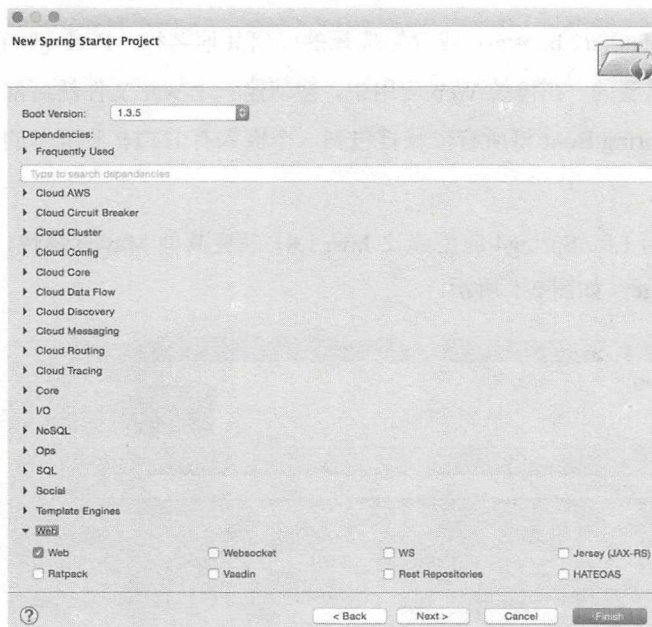


图 2-6

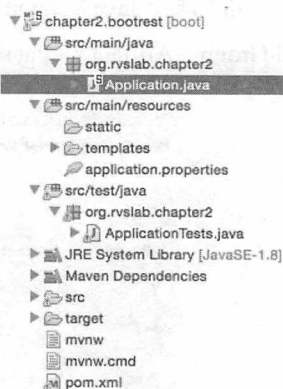


图 2-7

(7) 查看一下生成的应用。需要关注的文件如下：

- pom.xml
- Application.java
- Application.properties
- ApplicationTests.java

检查 POM 文件

parent 元素是 pom.xml 文件中需要关注的方面之一，如下：

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>1.3.4.RELEASE</version>
</parent>
```

spring-boot-starter-parent 模式是一个物料清单 (BOM)，是 Maven 的依赖关系管理使用的模式。BOM 是一种特殊的 POM 文件，用于管理项目所需的不同库版本。

使用 spring-boot-starter-parent POM 文件的优点是开发人员不必担心找不到适合各种库的兼容版本，如 Spring、Jersey、JUnit、Logback、Hibernate、Jackson 等。在我们的第一个传统示例中，添加了一个特定版本的 Jackson 库与 Spring4 协同工作。在本例中，这些由 spring-boot-starter-parent 模式负责。

启动器 POM 文件具有 Maven 构建所需的 Boot 依赖、资源过滤和插件配置。

提示

参考 <https://github.com/spring-projects/spring-boot/blob/1.3.x/spring-boot-dependencies/pom.xml> 查看启动器 parent(1.3.x 版本)中提供的不同依赖。如果需要，所有这些依赖都可以重写。

启动器 POM 文件本身不会将 JAR 依赖项添加到项目中。相反，它只是添加库版本。随后，当依赖关系添加到 POM 文件时，它们引用来自此 POM 文件的库版本。一些版本属性如下所示。

```
<spring-boot.version>1.3.5.BUILD-SNAPSHOT</spring-boot.version>
<hibernate.version>4.3.11.Final</hibernate.version>
<jackson.version>2.6.6</jackson.version>
<jersey.version>2.22.2</jersey.version>
<logback.version>1.1.7</logback.version>
<spring.version>4.2.6.RELEASE</spring.version>
<spring-data-releasetrain.version>Gosling-SR4</spring-data-releasetrain.version>
<tomcat.version>8.0.33</tomcat.version>
```

查看依赖关系部分，可以看到这是一个只有两个依赖关系的简单 POM 文件，如下所示。

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
```


Spring 微服务

```
</dependency>
</dependencies>
```

当选择 Web 时, spring-boot-starter-web 添加 Spring MVC 项目所需的所有依赖项, 包括作为嵌入式 HTTP 侦听器的 Tomcat 的依赖关系。这提供了一种有效的方式来获取作为单个包所需的所有依赖关系。单独的依赖可以替换为其他库, 如用 Jetty 替换 Tomcat。

与 Web 类似, Spring Boot 提供了一些 spring-boot-starter- *库, 如 amqp、aop、batch、data-jpa、thymeleaf 等。

对 pom.xml 文件进行审查的最后一项是 Java 8 属性。默认情况下, 父 POM 文件添加的 Java 版本是 Java 6。对于 Spring, 建议将 Java 版本覆盖为 8:

```
<java.version>1.8</java.version>
```

检查 Application.java

Spring Boot 默认在 src/main/java 下生成一个 org.rvslab.chapter2.Application.java 类用于启动, 如下:

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

在 Application 中只有一个 main 方法, 它将根据 Java 约定在启动时调用。main 方法通过调用 SpringApplication 中的 run 方法来启动 Spring Boot 应用。Application.class 作为参数传入, 告诉 Spring Boot 这是主要组件。这一切是由 @SpringBootApplication 注解完成的。@SpringBootApplication 是最高层注解, 它包括了下面的 3 个注解。

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
```

@Configuration 注解暗示包含的类声明了一个或多个 @Bean 注释。@Configuration 注解也是 @Component 的元注解, 被这个注解标示的类会被扫描到。

`@EnableAutoConfiguration` 注解告诉 Spring Boot 基于类路径中的可用依赖关系自动配置 Spring 应用程序。

检查 application.properties

默认 application.properties 文件位于 src/main/resources 下面。这是一个非常重要的文件，它为 Spring Boot 应用配置任何需要的属性。目前，这个文件是空的，本章后面的章节中会为配置文件添加内容。

检查 ApplicationTests.java

最后一个需要检查的文件是 src/test/java 下面的 ApplicationTests.java。这个类是 Spring Boot 应用的测试用例。

要实现第一个 RESTful 服务并添加 REST 终端，如下：

(1) 编辑 src/main/java 下面的 Application.java，添加一个 RESTful 服务实现。RESTful 服务和之前的例子完全相同。在 Application.java 文件尾部追加如下代码：

```
@RestController
class GreetingController{
    @RequestMapping("/")
    Greet greet() {
        return new Greet("Hello World!");
    }
}

class Greet {
    private String message;
    public Greet(){}
    public Greet(String message) {
        this.message = message;
    }
    // add getter and setter
}
```

(2) 运行，单击菜单项 Run As|Spring Boot App。tomcat 会在 8080 端口启动：

如图 2-8 所示，我们从日志中可以注意到：

UI、Paw 等。在这个例子中，Spring Boot 生成的默认测试类将会用来测试这个服务。

添加一个新的测试类到 Application.java，结果如下：

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes=Application.class)
@WebIntegrationTest
public class ApplicationTests {
    @Test
    public void testVanillaService() {
        RestTemplate restTemplate = new RestTemplate();
        Greet greet = restTemplate.getForObject("http://localhost:8080", Greet.class);
        Assert.assertEquals("Hello World!", greet.getMessage());
    }
}
```

注意：在 class 层新增了 `@WebIntegrationTest` 而移除了 `@WebAppConfiguration`。`@WebIntegrationTest` 注解十分方便，确保测试是针对一个完全正常运行的服务器触发的。也就是说，`@WebAppConfiguration` 和 `@IntegrationTest` 的组合将给出相同的结果。

还要注意，`RestTemplate` 用于调用 RESTful 服务。`RestTemplate` 是一个工具类，用于抽象 HTTP 客户端较低级别的详细信息。

要对上面的方法进行测试，可以打开一个终端窗口，进入项目目录，并运行 `mvn install`。

使用 Spring Initializr 开发 Spring Boot 微服务——HATEOAS 例子

在接下来的例子中，会用 Spring Initializr 创建一个 Spring Boot 微服务项目。Spring Initializr 是 STS 项目向导的简易替代，并提供了一个 Web UI 来配置和生成 Spring Boot 项目。Spring Initializr 的一个优势是可以通过网站生成能导入到任何 IDE 中的项目。

在本示例中，我们会探索用于基于 REST 的服务和 HAL（超文本应用程序语言）浏览器的 HATEOAS（作为应用程序状态引擎的超文本的缩写）的概念。

HATEOAS 是一种 REST 服务模式，其中导航链接作为响应内容的一部分提供。客户端应用程序检测这些状态并可以通过状态中的链接来触发转换动作。该方法在

响应移动和网络应用中特别有用，它的客户端基于用户导航模式来下载附加数据。

HAL 浏览器是一个方便的用于 hal + json 数据的 API 浏览器。HAL 是一种基于 JSON 的格式，它建立约定来表示资源之间的超链接。HAL 有助于我们了解 API 的情况。

提示

此示例的完整源代码可在本书代码文件的 chapter2.boothateoas 项目中查看。

使用 Spring Initializr 开发 HATEOAS 样例的具体步骤如下：

(1) 要访问 Spring Initializr，如图 2-10 所示，在浏览器上打开 <https://start.spring.io>。



图 2-10

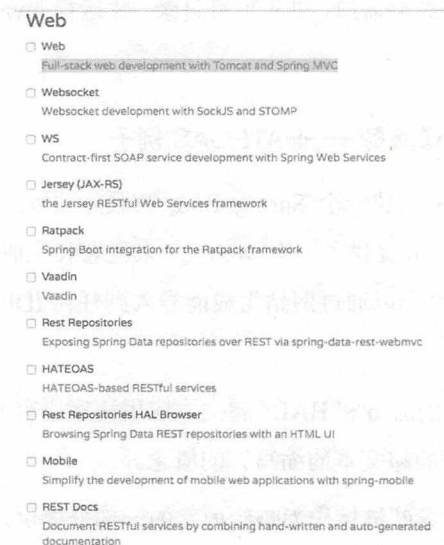


图 2-11

(2) 填写详细内容，如果是一个 Maven 工程，和之前一样，填写 Spring Boot version、group 和 artifact ID，并单击 Generate Project 按钮下面的 Switch to the full version。选择 Web、HATEOAS 和 Rest Repositories HAL Browser，如图 2-11 所示。保证 Java 版本是 8，而且包类型选择的是 JAR。

(3) 选择完成，单击 Generate Project 按钮。此时会生成一个 Maven 工程，将这个工程保存为一个 ZIP 文件存到浏览器的下载目录。

- (4) 解压文件并保存至您选择的目录下。
- (5) 打开 STS，进入 File 菜单并单击 Import。
- (6) 导航栏选择 Maven|Existing Maven Project 并单击 Next。

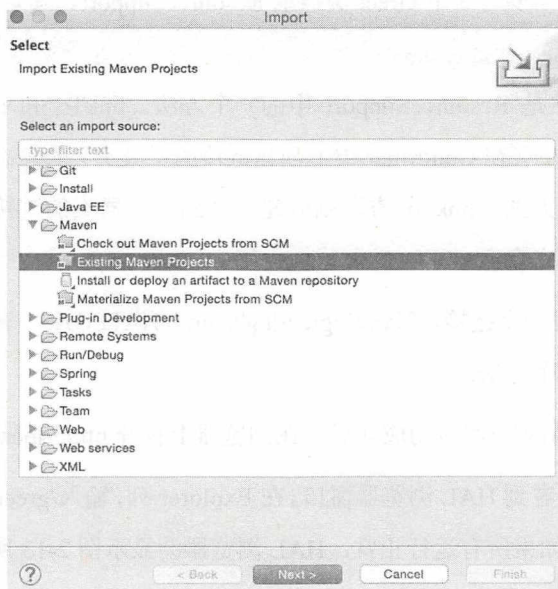


图 2-12

(7) 单击 Root Directory 下面的 Browse 并选择解压的目录，单击 Finish，这样 Maven 工程会被加载到 STS 的 Project Explorer。

- (8) 编辑 Application.java 文件，新增一个 REST 终端，如下：

```
@RequestMapping("/greeting")
@ResponseBody
public HttpEntity<Greet> greeting(@RequestParam(value="name",required=false,default Value=
"HATEOAS") String name) {
    Greet greet = new Greet("Hello"+ name);
    greet.add(linkTo(methodOn(GreetingController.class).greeting(name)).withSelfRel());
    return new ResponseEntity<Greet>(greet, HttpStatus.OK);
}
```

- (9) 注意这个 GreetingController 类和前一个例子中基本相同。然而，这一次添

加了一个名为 `greeting` 的方法。在这个新的方法中，定义了一个默认为 HATEOAS 的额外可选请求参数。如下代码添加了一个到结果 JSON 代码的链接。

```
greet.add(linkTo(methodOn(GreetingController.class).greeting(name)).withSelfRel());
```

下一步，我们需要写一个 `Greet` 类继承 `ResourceSupport`。其余代码保持相同：

```
class Greet extends ResourceSupport{
```

(10) `add` 方法是 `ResourceSupport` 中的一个方法。`linkTo` 和 `methodOn` 方法是 `ControllerLinkBuilder`（在 `Controller` 类上创建链接的工具类）的静态方法。`methodOn` 方法调用一个虚拟方法，`linkTo` 方法则创建一个到控制器类的链接。这里我们使用 `withSelfRel` 来指向它自己。

(11) 此时生成一个链接，默认是 `/greeting?name=HATEOAS`。客户端可以读取这个链接并调用另一个方法。

(12) Spring Boot 应用启动成功后，在浏览器上打开 `http://localhost:8080`。

(13) 此时你会看到 HAL 浏览器窗口。在 Explorer 列，输入 `/greeting?name=World!` 并单击 Go 按钮。如果所有运行正常，HAL 浏览器会显示图 2-13 所示的响应明细。

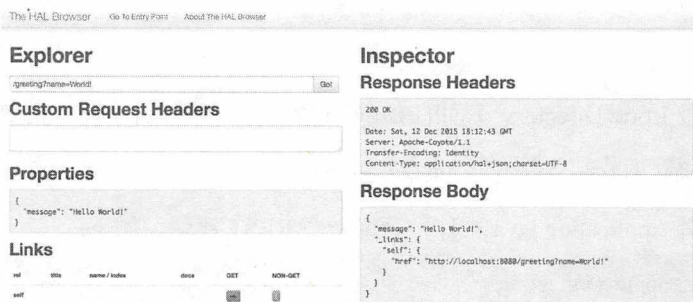



图 2-13

如图 2-13 所示，`Response Body` 部分有指回相同服务的 `href` 链接的结果。这是因为我们将引用指向它自己。也可以查看 `Links` 部分。`Self` 行里面的  是导航链接。

在这个简单的例子中读者可能无法体会 HAL 窗口的便捷，但是在具有许多相关实体的更大的应用程序中，HAL 窗口会带来很多方便。客户端可以使用提供的链接轻松地在这些实体之间来回导航。

下一步是什么

到目前为止已经回顾了一些基本的 Spring Boot 例子。这一章剩下的部分我们会讨论一些对微服务开发至关重要的 Spring Boot 特性。在下一节，我们会学习如何使用动态配置属性、更改默认嵌入的 Web 服务器和微服务安全及处理微服务时的跨源实现。

提示

这个例子的所有源代码可以在本书代码文件的 `chapter2.boot-advanced` 工程中获取。

Spring Boot 配置

本节我们会关注 Spring Boot 的配置层。通过修改之前开发的 `chapter2.bootrest`，来展示 Spring Boot 的配置功能。将 `chapter2.bootrest` 保存一份副本，重命名为 `chapter2.boot-advanced`。

理解 Spring Boot 自动配置

Spring Boot 通过扫描类路径中可用的依赖库来履行“约定优于配置”的原则。对 POM 文件中的每一个 `spring-boot-starter-*` 依赖，Spring Boot 执行一个默认的 `AutoConfiguration` 类。`AutoConfiguration` 类使用 `*AutoConfiguration` 词汇匹配模式，其中的 `*` 表示库。例如，JPA 仓库的自动配置是通过 `JpaRepositoriesAutoConfiguration` 来实现的。

用 `--debug` 参数运行应用来查看自动配置的日志报告。如下命令显示了 `chapter2.boot-advanced` 项目自动配置的日志报告：

```
$java -jar target/bootadvanced-0.0.1-SNAPSHOT.jar --debug
```

这是自动配置类的一些例子：

- ServerPropertiesAutoConfiguration
- RepositoryRestMvcAutoConfiguration
- JpaRepositoriesAutoConfiguration
- JmsAutoConfiguration

如果应用有特殊需求，而且您想要完全采用自己的配置，可以指定排除哪些库的依赖。下面是排除 `DataSourceAutoConfiguration` 的例子：

```
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
```

覆盖默认配置值

也可以使用 `application.properties` 文件来覆盖默认配置值。STS 为 `application.properties` 提供了一个易于自动完成的上下文帮助，如图 2-14 所示。

```
server.port 9090

spring.j
spring.jackson.date-format : String
spring.jackson.deserialization : Map<com.fasterxml.jackson.databind.DeserializationFeatu
spring.jackson.generator : Map<com.fasterxml.jackson.core.JsonGenerator.Feature[AUTO
spring.jackson.joda-date-time-format : String
spring.jackson.locale : Locale
spring.jackson.mapper : Map<com.fasterxml.jackson.databind.MapperFeature[USE_ANNOC
spring.jackson.parser : Map<com.fasterxml.jackson.core.JsonParser.Feature[AUTO_CLOS
spring.jackson.property-naming-strategy : String
spring.jackson.serialization : Map<com.fasterxml.jackson.databind.SerializationFeature[W
spring.jackson.serialization-inclusion : com.fasterxml.jackson.annotation.JsonInclude$Incl
spring.jackson.time-zone : TimeZone
spring.jersey.application-path : String
spring.jersey.filter.order : int
spring.jersey.init : Map<String, String>
spring.jersey.type : org.springframework.boot.autoconfigure.jersey.JerseyProperties$TypeP
spring.jpa.hibernate.name : String
```

图 2-14

在图 2-14 中，`server.port` 可以设置成 9090。将应用重启，会运行在 9090 端口上。

改变配置文件路径

根据十二因素应用的准则，配置参数需要从代码中外置化。Spring Boot 将所有配置外部化到 `application.properties` 中。然而，这些配置文件仍然是编译过程的一部分。此外，可以通过设置如下属性来从包外部读取配置：

```
spring.config.name= # config file name
spring.config.location = # location of config file
```

这里，`spring.config.location` 可以是本地文件路径。

如下命令启动带有外部支持的配置文件的 Spring Boot 应用:

```
$java -jar target/bootadvanced-0.0.1-SNAPSHOT.jar --spring.config.name=bootrest.properties
```

读取自定义属性

在启动时, `SpringApplication` 加载所有属性并将它们添加至 `Spring` 的 `Environment` 类。将一个名为 `bootrest.customproperty` 的自定义属性添加到 `application.properties` 文件。将 `Spring` 的 `Environment` 类自动注入到 `GreetingController` 类。编辑 `GreetingController` 类以从 `Environment` 读取自定义属性, 并添加一个日志语句将定制属性打印到控制台。整体步骤如下所示。

(1) 添加下面的属性到 `application.properties` 文件:

```
bootrest.customproperty=hello
```

(2) 然后, 编辑 `GreetingController` 类:

```
@Autowired
Environment env;

Greet greet() {
    logger.info("bootrest.customproperty" + env.getProperty("bootrest.customproperty"));
    return new Greet("Hello World! ");
}
```

(3) 重新运行应用。能够在控制台看到下面的日志:

```
org.rvslab.chapter2.GreetingController:bootrest.customproperty hello
```

使用 .yaml 文件配置

作为 `application.properties` 的另一种选择, 可以使用 .yaml 文件。相比于扁平的属性文件, `YAML` 支持一个类似 `JSON` 结构的配置。

举个例子, 简单地用 `application.yaml` 替换 `application.properties` 并添加如下属性:

```
server
  port: 9080
```

重新启动应用来查看在控制台打印的端口号。

使用多配置文件

此外, 可以根据不同的环境 (如开发、测试、分段、生产等), 使用不同的配置

文件。可以为不同环境的相同属性配置不同的值。当对不同的环境运行 Spring Boot 应用程序时，这是非常方便的。从一个环境移动到另一个环境时，不需要重建。

编辑.yaml 文件，如下所示。Spring Boot 基于虚线分隔符组织配置文件属性：

```
spring:
  profiles: development
server:
  port: 9090
---
spring:
  profiles: production
server:
  port: 8080
```

运行 Spring Boot 应用程序，并查看配置文件的使用：

```
mvn -Dspring.profiles.active=production install
mvn -Dspring.profiles.active=development install
```

活跃配置文件可以在代码中使用 `@ActiveProfiles` 注解指定，这在运行测试用例时尤其方便，如下所示：

```
@ActiveProfiles("test")
```

读取属性的其他选项

属性可以通过多种方式加载：

- 命令行参数（`-Dhost.port=9090`）。
- 操作系统环境变量 JNDI (`java:comp/env`)。

修改默认嵌入的 Web 服务器

嵌入的 HTTP 监听器可以轻松地根据需要自定义。Spring Boot 默认支持 Tomcat、Jetty 和 Undertow。在下面的例子中，使用 Undertow 替换 Tomcat：

```
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
<exclusions>
  <exclusion>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
  </exclusion>
</exclusions>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-undertow</artifactId>
</dependency>
```

实现 Spring Boot 安全性

微服务的安全性是非常重要的。本节将使用 chapter2.bootrest 来回顾一些用于保护 Spring Boot 微服务的基本措施，以演示安全特性。

使用基础的安全措施保护微服务

给 Spring Boot 添加基本认证是非常简单的。添加如下依赖至 pom.xml 以包含必须的 Spring 安全库文件：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

打开 Application.java 并添加@EnableGlobalMethodSecurity 至 Application 类。这个注解会开启方法层的安全性：

```
@EnableGlobalMethodSecurity
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
```



```

        SpringApplication.run(Application.class, args);
    }
}

```

默认基本认证假设用户名是 `user`。默认密码启动时会在控制台打印出来。或者将用户名和密码添加到 `application.properties`，如下：

```

@Test
public void testSecureService() {
    String plainCreds = "guest:guest123";
    HttpHeaders headers = new HttpHeaders();
    headers.add("Authorization", "Basic " + new String(Base64.encode(plainCreds.getBytes())));
    HttpEntity<String> request = new HttpEntity<String>(headers);
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<Greet> response = restTemplate.exchange("http://localhost:8080",
        HttpMethod.GET, request, Greet.class);
    Assert.assertEquals("Hello World!",
        response.getBody().getMessage());
}

```

如代码中所示，将创建一个带有 Base64 编码的 `username-password` 字符串的新的认证请求头。

使用 Maven 重新运行应用。新的测试用例会通过，但老的测试用例会有异常并失败。这是因为早期的测试用例现在运行时没有证书，因此服务器拒绝请求并显示：

```
org.springframework.web.client.HttpClientErrorException: 401 Unauthorized
```

使用 OAuth2 保护微服务

本节我们将了解 OAuth2 的基本 Spring Boot 配置。当客户端应用程序需要访问受保护资源时，客户端向授权服务器发送请求。授权服务器验证请求并提供访问令牌。此访问令牌针对每个客户端到服务器请求进行验证。发送的请求和响应取决于授予类型。

提示

获取更多关于 OAuth 和授权类型，请访问：<http://oauth.net>。

在这个例子中，我们使用资源所有者密码凭据授权方法。

如图 2-15 所示，资源所有者向客户端提供用户名和密码。然后，客户端通过提供凭证信息向授权服务器发送令牌请求。授权服务器授权客户端并返回一个访问令牌。在每个后续请求中，服务器都要验证客户端令牌。

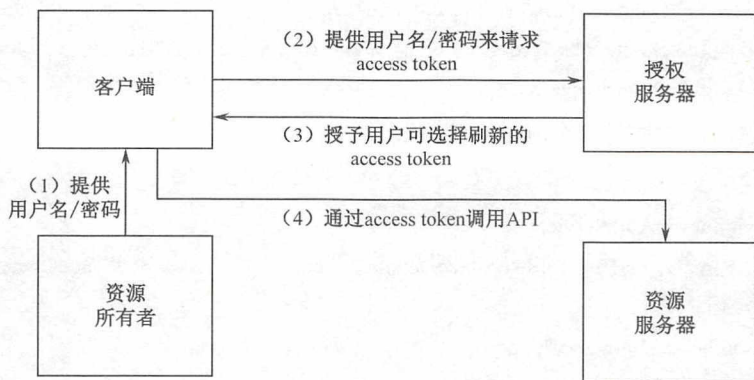


图 2-15

要在我们的例子中实现 OAuth2，需要执行如下步骤：

(1) 第一步，更新 pom.xml 里的 OAuth2 依赖，如下：

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
  <version>2.0.9.RELEASE</version>
</dependency>
```

(2) 接下来，添加两个新注解到 Application.java 文件上，@EnableAuthorizationServer 和 @EnableResourceServer。@EnableAuthorizationServer 注解创建了一个带有内存库的服务器，存储客户端指令并提供客户端的用户名、密码、客户端 ID 和密钥。@EnableResourceServer 注解用于访问令牌。这将启用 spring 安全过滤器，通过传入的 OAuth2 令牌进行身份验证。在我们的示例中，授权服务器和资源服务器是相同的。然而，在实践中，这两个服务器是独立运行的。代码如下：

```
@EnableResourceServer
@EnableAuthorizationServer
@SpringBootApplication
```

```
public class Application {
```

(3) 在 `application.properties` 文件中添加如下属性:

```
security.user.name=guest
security.user.password=guest123
security.oauth2.client.clientId: trustedclient
security.oauth2.client.clientSecret: trustedclient123
security.oauth2.client.authorized-grant-types: authorization_code,refresh_token,password
security.oauth2.client.scope: openid
```

(4) 然后, 添加另一个测试用例来测试 OAuth2:

```
@Test
public void testOAuthService() {
    ResourceOwnerPasswordResourceDetails resource = new ResourceOwnerPassword
ResourceDetails();
    resource.setUsername("guest");
    resource.setPassword("guest123");
    resource.setAccessTokenUri("http://localhost:8080/oauth/token"
);
    resource.setClientId("trustedclient");
    resource.setClientSecret("trustedclient123");
    resource.setGrantType("password");
    DefaultOAuth2ClientContext clientContext = new DefaultOAuth2ClientContext();
    OAuth2RestTemplate restTemplate = new OAuth2RestTemplate(resource, clientContext);
    Greet greet = restTemplate.getForObject("http://localhost:8080", Greet.class);
    Assert.assertEquals("Hello World!", greet.getMessage());
}
```

如上面的代码所示, 通过将详细资源信息封装在对象中来创建特殊的 REST 模板 `OAuth2RestTemplate`。 `OAuth2RestTemplate` 处理下面的 OAuth2 进程。

(5) 使用 `mvn install` 重新运行应用。前面两个测试用例会失败, 新的一个会成功。这是因为服务器只接受开启 OAuth2 的请求。

这些是由 Spring Boot 原生提供的快速配置, 但是不能够满足生产的需求。我们可能需要自定义 `ResourceServerConfigurer` 和 `AuthorizationServerConfigurer` 以使其可用于生产环境。

为微服务开启跨域访问

当客户端 Web 应用程序从一个来源请求来自另一个来源的数据时，浏览器通常受到限制。开启跨域访问通常被称为 CORS（Cross-Origin Resource Sharing）。

图 2-16 显示了如何启用跨域请求。由于每个微服务都运行自己的域，它很容易陷入客户端 Web 应用程序消费多个数据源的问题。例如，浏览器客户端同时从客户微服务访问客户和从订单微服务访问订单历史数据，这种情况在微服务世界中是很常见的。

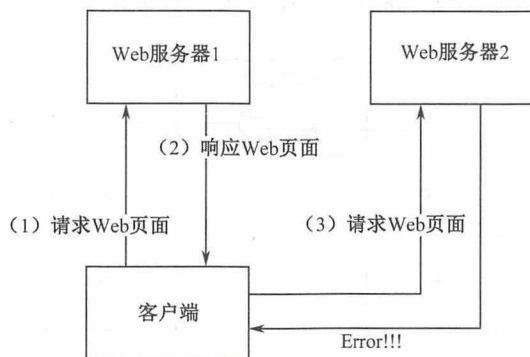


图 2-16

Spring Boot 提供了一种简单的声明方法来启用跨域请求，如下所示：

```
@RestController
class GreetingController{
    @CrossOrigin
    @RequestMapping("/")
    Greet greet(){
return new Greet("Hello World!");
    }
}
```

默认情况下接收所有域和请求头。我们可以通过给指定域的访问权限来进一步自定义跨域注解。例如，`@CrossOrigin` 注解开启一个方法或类来接收跨域请求：


```
@CrossOrigin("http://mytrustedorigin.com")
```

可以使用 `WebMvcConfigurer` bean 并定制 `addCorsMappings(CorsRegistry 注册表)` 方法来启用全局 CORS。

实现 Spring Boot 通知

理想情况下使用发布—订阅模式，微服务的所有交互应该都是异步的。Spring Boot 提供了一种通过配置进行无障碍的消息传递的方案。

如图 2-17 所示，在这个例子中，我们将创建一个具有发送方和接收方的 Spring Boot 应用程序，两者都通过外部队列连接。执行下列步骤：

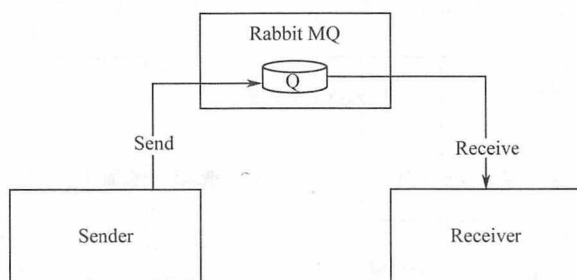


图 2-17

提示

此示例的完整源代码可在本书代码文件中的 `chapter2.bootmessaging` 项目中获取。

(1) 使用 STS 创建一个新项目。在此示例中，不是选择 Web，而是在 I/O 下选择 AMQP，如图 2-18 所示。

(2) 此示例还将需要 Rabbit MQ。请从 <https://www.rabbitmq.com/download.html> 下载并安装最新版本的 Rabbit MQ。本书中使用 Rabbit MQ 3.5.6。

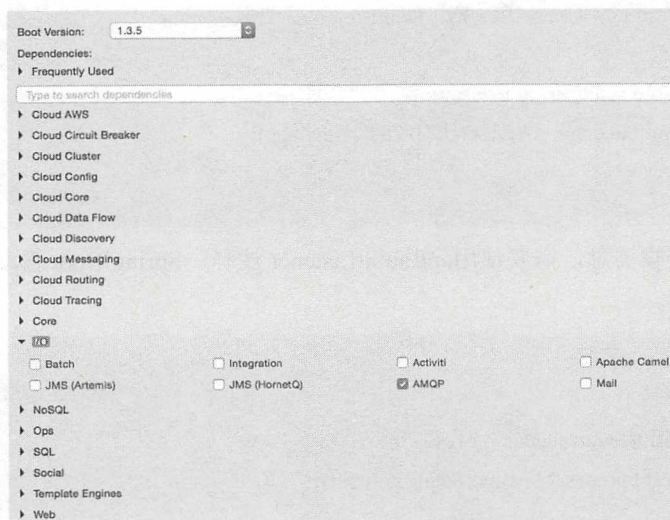


图 2-18

(3) 按照网站上记录的步骤安装。准备好后，启动 RabbitMQ 服务器：

```
$.rabbitmq-server
```

(4) 修改 application.properties 文件的配置，指向 RabbitMQ。以下配置使用 RabbitMQ 的默认端口、用户名和密码：

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

(5) 将一个消息发送器组件和一个名为 TestQ 的 org.springframework.amqp.core.Queue 类型的队列添加到 src/main/java 下的 Application.java 文件中。RabbitMessagingTemplate 是一种方便的发送消息的方法，它抽象所有的消息语义。Spring Boot 提供了发送消息的样板配置：

```
@Component
class Sender {
    @Autowired
    RabbitMessagingTemplate template;

    @Bean
    Queue queue() {
```

```

        return new Queue("TestQ", false);
    }
    public void send(String message){
        template.convertAndSend("TestQ", message);
    }
}

```

(6) 要接收消息，需要使用 `@RabbitListener` 注释。Spring Boot 自动配置所有必需的样板配置：

```

@Component
class Receiver {
    @RabbitListener(queues = "TestQ")
    public void processMessage(String content) {
        System.out.println(content);
    }
}

```

(7) 最后一步是将发送器连接到我们的主应用程序，并实现 `CommandLineRunner` 的运行方法以发起消息发送。当应用程序初始化时，它调用 `CommandLineRunner` 的 `run` 方法，如下所示：

```

@SpringBootApplication
public class Application implements CommandLineRunner{
    @Autowired
    Sender sender;
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
    @Override
    public void run(String... args) throws Exception {
        sender.send("Hello Messaging..!!!");
    }
}

```

(8) 将应用程序作为 Spring Boot 应用程序运行并验证输出。控制台会显示如下日志：

```

Hello Messageing..!!!

```

开发全面的微服务示例

到目前为止，我们所考虑的例子不过是一个简单的“Hello world”。综合我们所学到的，本节将介绍具有业务逻辑和原始数据存储的 Customer Profile 微服务的端到端的一种实现，从而展示不同微服务之间的交互。

如图 2-19，在本示例中，将开发两个微服务——客户配置文件和客户通知服务：

如图 2-19 所示，Customer Profile 微服务公开了创建、读取、更新及删除（CRUD）客户信息和注册服务的方法，从而实现登记客户账户信息。注册过程应用特定的业务逻辑，保存客户详细信息，并向 Customer Notification 微服务发送消息。Customer Notification 微服务接受注册服务发送的消息，并使用 SMTP 服务器向客户发送电子邮件。异步消息传递用于集成 Customer Profile 与 Customer Notification 微服务。

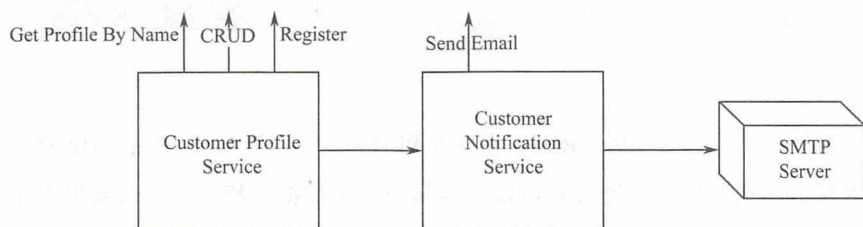


图 2-19

Customer 微服务类域模型图如图 2-20 所示。

图 2-20 中的 CustomerController 是 REST 端点，它调用组件类 Customer Component。组件类/bean 处理所有的业务逻辑。CustomerRepository 是一个 Spring 数据 JPA 存储库，定义为处理 Customer 实体的持久性。

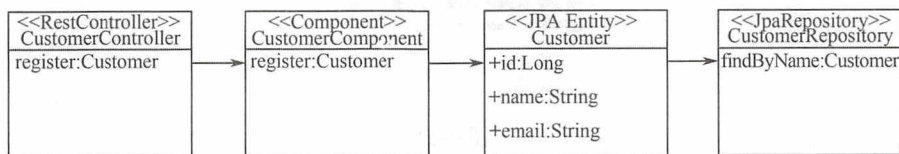


图 2-20

提示

此示例的完整源代码可在本书代码文件中的 chapter2.bootcustomer 项目中获取。

(1) 创建一个新的 Spring Boot 项目，并将其命名为 chapter2.bootcustomer，方式与之前相同，按照图 2-22 所示启动选项。



图 2-21

这将创建一个具有 JPA、REST 存储库和 H2 数据库的 Web 项目。H2 是一个微小的内存嵌入式数据库，通过它可以很容易地展示数据库特性。在现实世界中，建议使用适当的企业级数据库。此示例使用 JPA 来定义持久性实体，并使用 REST 存储库来公开基于 REST 的存储库服务。项目结构如图 2-22 所示。

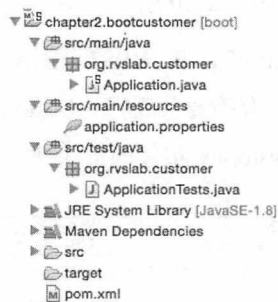


图 2-22

(2) 添加名为 Customer 的实体类来开始构建应用程序。为了简单起见，这个类只有 3 个属性：自动生成的 ID 字段、名称和电子邮件。

```
@Entity
class Customer {
```



```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
private String name;
private String email;
```

(3) 添加一个存储库类来对 `Customer` 进行持久化处理。`CustomerRepository` 扩展了标准 JPA 存储库。这意味着所有 CRUD 方法和默认 finder 方法都是由 Spring Data JPA 仓库自动实现的，如下所示：

```
@RepositoryRestResource
interface CustomerRespository extends JpaRepository<Customer,Long>{
    Optional<Customer> findByName(@Param("name") String name);
}
```

在此示例中，我们向存储库类 `findByName` 添加了一个新方法，它根据客户名称搜索客户，并在匹配到结果时返回客户对象。

(4) `@RepositoryRestResource` 注解通过 RESTful 服务访问存储库。默认情况下，也会同时启用 HATEOAS 和 HAL。对于 CRUD 方法，不需要额外的业务逻辑，我们将保留它，因为它没有控制器或组件类。使用 HATEOAS 帮助我们轻松浏览客户存储库。注意：我们没有配置数据库地址。因为 H2 库在类路径中，所有配置都是由 Spring Boot 根据 H2 自动配置默认完成的。

(5) 添加 `CommandLineRunner` 来更新 `Application.java` 文件，以使用一些客户记录来初始化存储库，如下所示：

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
    @Bean
    CommandLineRunner init(CustomerRespository repo) {
        return (evt) -> {
            repo.save(new Customer("Adam","adam@boot.com"));
            repo.save(new Customer("John","john@boot.com"));
            repo.save(new Customer("Smith","smith@boot.com"));
        };
    }
}
```

```

repo.save(new Customer("Edgar","edgar@boot.com"));
repo.save(new Customer("Martin","martin@boot.com"));
repo.save(new Customer("Tom","tom@boot.com"));
repo.save(new Customer("Sean","sean@boot.com"));

};

}

}

```

(6) CommandLineRunner，注解为一个 bean，指示它应该在 SpringApplication 启动时将 6 个示例客户记录插入数据库。

(7) 此时，将应用程序作为 Spring Boot App 运行。打开 HAL 浏览器，跳转到 <http://localhost:8080>。

(8) 在“资源管理器”中，指向 <http://localhost:8080/customers>，然后单击 Go，会列出 HAL 浏览器的“响应主体”部分中的所有客户。

(9) 在“资源管理器”中，输入 <http://localhost:8080/customers?size=2&page=1&sort=name>，然后单击 Go，会自动对存储库执行分页和排序并返回结果。

当页面大小设置为 2 并且请求第一页时，它将以排序顺序返回两个记录。

(10) 查看“链接”部分。如图 2-23 所示，可以方便地跳转到 first、next、prev 和 last。这些是使用由存储库浏览器自动生成的 HATEOAS 链接完成的。

Links

rel	title	name / index	docs	GET	NON-GET
first				→	!
prev				→	!
self				→	!
next				→	!
last				→	!
profile				→	!
search				→	!

图 2-23

(11) 此外，可以通过选择适当的链接（如 <http://localhost:8080/customers/2>）来了解客户的详细信息。

(12) 下一步，添加一个控制器类 `CustomerController` 来处理服务端点。这个类只有一个端点，`/register` 用于注册一个客户。如果成功，它返回 `Customer` 对象作为响应，如下所示：

```
@RestController
class CustomerController{
    @Autowired
    CustomerRegistrar customerRegistrar;
    @RequestMapping( path="/register", method = RequestMethod.POST)
    Customer register(@RequestBody Customer customer){
        return customerRegistrar.register(customer);
    }
}
```

(13) 添加一个 `CustomerRegistrar` 组件来处理业务逻辑。这样组件中只有最少的业务逻辑。在这个组件类中，在注册客户时，我们将检查客户名称是否已经存在于数据库中。如果它不存在，那么我们将插入一个新记录；否则，我们将发送一条错误消息，如下所示：

```
@Component
class CustomerRegistrar {
    CustomerRespository customerRespository;
    @Autowired
    CustomerRegistrar(CustomerRespository customerRespository){
        this.customerRespository = customerRespository;
    }
    Customer register(Customer customer){
        Optional<Customer> existingCustomer = customerRespository.findByName(customer.
getName());
        if (existingCustomer.isPresent()){
            throw new RuntimeException("is already exists");
        } else {
            customerRespository.save(customer);
        }
    }
}
```

```
        return customer;
    }
}
```

(14) 重新启动引导应用程序并使用 HAL 浏览器通过 URL `http://localhost:8080` 进行测试。

(15) 将资源管理器字段指向 `http://localhost:8080/customers`。在“链接”部分查看结果，如图 2-24 所示。

Links

rel	title	name / index	docs	GET	NON-GET
self				→	!
profile				→	! Perform non-GET rec
search				→	!

图 2-24

(16) 单击 self 中的 NON-GET 选项，会打开一个表单来创建一个新客户，如图 2-25 所示。

Create/Update

Customer

Name

World

Email

world@hello.com

Action:

POST

http://localhost:8080/register

Make Request

图 2-25

(17) 填写表单并更改操作，单击图 2-25 上的“MakeRequest”按钮。调用注册服务并注册客户。请试着用相同的名字再注册一次，作为注册失败的测试用例。

(18) 让我们通过集成客户通知服务来通知客户完成示例的最后一部分。注册成功后，通过异步调用客户通知微服务向客户发送电子邮件。

(19) 首先更新 CustomerRegistrar，通过消息传递调用第二个微服务。我们注入了一个 Sender 组件，通过将客户的电子邮件地址传递给发件人来向客户发送通知，如下所示：

```
@Component
@Lazy
class CustomerRegistrar {
    CustomerRespository customerRespository;
    Sender sender;
    @Autowired
    CustomerRegistrar(CustomerRespository customerRespository, Sender sender){
        this.customerRespository = customerRespository;
        this.sender = sender;
    }
    Customer register(Customer customer){
        Optional<Customer> existingCustomer = customerRespository.findByName(customer.
getName());
        if (existingCustomer.isPresent()){
            throw new RuntimeException("is already exists");
        } else {
            customerRespository.save(customer);
            sender.send(customer.getEmail());
        }
        return customer;
    }
}
```

(20) 发送方组件将基于 RabbitMQ 和 AMQP。传递最后一个消息时，会使用 RabbitMessagingTemplate:

```
@Component
@Lazy
```

```
class Sender {  
    @Autowired  
    RabbitMessagingTemplate template;  
  
    @Bean  
    Queue queue() {  
        new Queue("CustomerQ", false);  
    }  
  
    public void send(String message){  
        template.convertAndSend("CustomerQ", message);  
    }  
}
```

@Lazy 注释有助于加快启动速度。这些 bean 只有在需要时才被初始化。

(21) 更新 application.property 文件以包括 Rabbit MQ 相关属性，如下所示：

```
spring.rabbitmq.host=localhost
```

```
spring.rabbitmq.port=5672
```

```
spring.rabbitmq.username=guest
```

```
spring.rabbitmq.password=guest
```

(22) 下一步是发送消息。为了使用消息和发送电子邮件，我们将创建一个通知服务。为此，我们创建另一个 Spring Boot 服务，chapter2.bootcustomnotification。确保在创建 Spring Boot 服务时选择 AMQP 和 Mail 启动库。AMQP 和邮件都在 I/O 下。

(23) hapter2.bootcustomnotification 项目的包结构如图 2-26 所示。



图 2-26

(24) 添加 Receiver 类。Receiver 类接收由客户配置文件服务发送的客户消息。在消息到达时，它发送电子邮件，如下所示：

```
@Component
class Receiver {
    @Autowired
    Mailer mailer;
    @Bean
    Queue queue() {
        return new Queue("CustomerQ", false);
    }
    @RabbitListener(queues = "CustomerQ")
    public void processMessage(String email) {
        System.out.println(email);
        mailer.sendMail(email);
    }
}
```

(25) 添加 JavaMailSender 来向客户发送电子邮件：

```
@Component
class Mailer {
    @Autowired
    private JavaMailSender javaMailService;
    public void sendMail(String email){
        SimpleMailMessage mailMessage=new SimpleMailMessage();
        mailMessage.setTo(email);
        mailMessage.setSubject("Registration");
        mailMessage.setText("Successfully Registered");
        javaMailService.send(mailMessage);
    }
}
```

在后台，Spring Boot 自动配置 JavaMailSender 所需的所有参数。

(26) 要测试 SMTP，需要进行相关设置，以确保邮件正在传出。在此示例中，需要使用 FakeSMTP。您可以从 <http://nilhcem.github.io/FakeSMTP> 下载 FakeSMTP。

(27) 下载 fakeSMTP-2.0.jar 之后，通过执行以下命令运行 SMTP 服务器：

```
$ java -jar fakeSMTP-2.0.jar
```

将打开一个 GUI 来监视电子邮件。单击侦听端口文本框旁边的“Start Server”按钮。

(28) 使用以下配置参数更新 `application.properties` 以连接到 RabbitMQ 及邮件服务器：

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
spring.mail.host=localhost
spring.mail.port=2525
```

(29) 我们准备好端到端测试我们的微服务。启动两个 Spring Boot 应用程序。通过 HAL 浏览器重复客户创建步骤。在提交请求后，我们将能够在 SMTP GUI 中看到电子邮件。

在内部，客户配置文件服务异步调用客户通知服务，后者又将电子邮件发送到 SMTP 服务器，如图 2-27 所示。

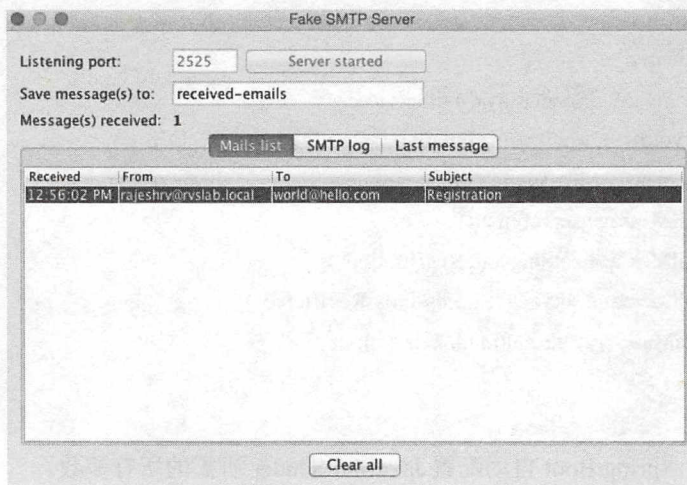


图 2-27

Spring Boot Actuator

前面的部分探讨了开发微服务所需的大多数 Spring Boot 特性。在本节中，将探讨 Spring Boot 在生产环境中的一些特性。

Spring Boot Actuator 提供了一个出色的开箱即用机制来监视和管理 Spring Boot 在生产中的应用。

提示

本示例的完整源代码可在本书代码文件中的 chapter2.Boot Actuator 项目中获取。

(1) 创建另一个 Spring Starter 项目并将其命名为 chapter2.Boot Actuator。这次，在 Ops 下选择 Web 和 Actuator。与 chapter2.bootrest 项目类似，使用 greet 方法添加一个 GreeterController 端点。

(2) 以 Spring Boot 启动应用。

(3) 将浏览器指向 localhost:8080/actuator，然后查看“链接”部分。

链接部分下有多个链接，这些由 Spring Boot Actuator 自动暴露，如图 2-28 所示。

Links

rel	title	name / index	docs	GET	NON-GET
self				➔	🔒
dump				➔	🔒
configprops				➔	🔒
env				➔	🔒
mappings				➔	🔒
info				➔	🔒
health				➔	🔒
autoconfig				➔	🔒
metrics				➔	🔒
trace				➔	🔒
beans				➔	🔒

图 2-28

其中的一些重要链接如下：

- dump：执行线程备份并显示结果。
- mappings：列出所有 HTTP 请求映射。
- info：显示有关应用程序的信息。
- health：显示应用程序的运行状况。
- autoconfig：显示自动配置报告。
- metrics：显示从应用程序收集的不同指标。

使用 Jconsole 监控

我们可以使用 JMX 控制台查看 Spring Boot 信息。从 JConsole 连接到远程 Spring Boot 实例，引导信息如图 2-29 所示。

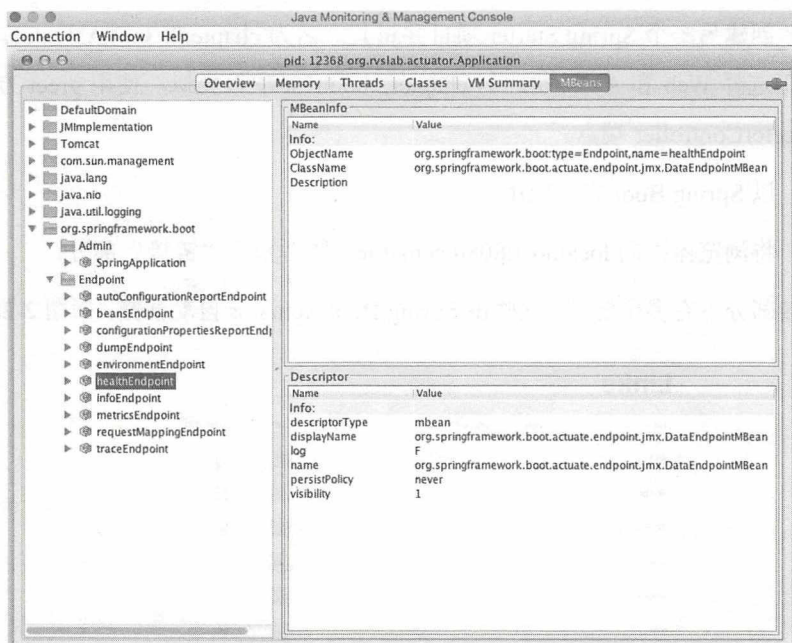


图 2-29

使用 SSH 监控

Spring Boot 提供使用 SSH 远程访问应用程序的功能。以下命令从终端窗口连接

到 Spring Boot 应用程序：

```
$ ssh -p 2000 user@localhost
```

可以通过在 `application.properties` 文件中添加 `shell.auth.simple.user.password` 属性来定制密码：

```
shell.auth.simple.user.password=admin
```

当使用前面的命令连接到 Boot 应用程序后，就可以访问 Boot Actuator 的信息了。以下是通过 CLI 访问度量信息的示例：

- `help`：它列出了所有可用的选项。
- `dashboard`：它显示了大量的系统信息。

配置应用信息

可以在 `application.properties` 中设置以下属性来自定义与应用程序相关的信息。添加后，重新启动服务器并访问 Actuator 的 `/info` 端点，以查看更新的信息：

```
info.app.name=Boot actuator  
info.app.description= My Greetings Service  
info.app.version=1.0.0
```

添加自定义运行状况模块

将新的自定义模块添加到 Spring Boot 应用程序并不那么复杂。为了演示此功能，假设一个服务在一分钟内获得多于两个事务，则服务器状态将设置为“不在服务”。

为了定制自定义模块，我们必须实现健康指示器接口并覆盖健康方法。以下是一个简单的例子实现：

```
class TPSCounter {  
    LongAdder count;  
    int threshold = 2;
```

```

Calendar expiry = null;
TPSCounter(){
    this.count = new LongAdder();
    this.expiry = Calendar.getInstance();
    this.expiry.add(Calendar.MINUTE, 1);
}
boolean isExpired(){
    return Calendar.getInstance().after(expiry);
}
boolean isWeak(){
    return (count.intValue() > threshold);
}
void increment(){
    count.increment();
}
}

```

前面的类是一个简单的 POJO 类，在窗口中维护事务计数。isWeak 方法检查特定窗口中的事务是否达到其阈值。isExpired 方法检查当前窗口是否过期。增量方法只是增加计数器值。

下一步，通过扩展 HealthIndicator 来实现我们的自定义健康指示器类 TPSHealth，如下所示：

```

@Component
class TPSHealth implements HealthIndicator {
    TPSCounter counter;

    @Override
    Public Health health(){
        boolean health = counter.isWeak(); // perform some specific health check
        if (health) {
            return Health.outOfService().withDetail("Too many
            requests", "OutOfService").build();
        }
        return Health.up().build();
    }
}

```

- void updateTx(){
 if(counter == null || counter.isExpired()){


```

        counter = new TPSCounter();
    }
    counter.increment();
}
}

```

health 方法检查计数器是否超出阈值。超出阈值意味着任务超过了服务的负载。如果超出，则将实例标记为“out of service”。

最后，我们将 TPS Health 自动连接到 Greeting Controller 类，然后在 greet 方法中调用 health.updateTx ()，如下所示：

```

Greet greet(){
    logger.info("Serving Request.....!!!");
    health.updateTx();
    return new Greet("Hello World!");
}

```

转到 HAL 浏览器中的 / health 终端节点，并查看服务器的状态。

现在，打开另一个浏览器，指向 <http://localhost:8080>，然后触发服务 2 次或 3 次。返回到 / health 端点并刷新以查看状态。它应该更改为停用。

在此示例中，由于除了收集健康状态之外没有采取任何操作，即使状态为“out of service”，新的服务调用仍将通过。然而，在现实世界中，程序应该读取 /health 并阻止请求被转发到这个实例。

构建自定义指标

与健康类似，度量的定制也是可以实现的。以下示例显示如何添加计数器服务和计量服务，仅用于演示目的：

```

@Autowired
CounterService counterService;

@Autowired
GaugeService gaugeService;

```

在 greet 方法中添加以下方法：

```

this.counterService.increment("greet.txnCount");
this.gaugeService.submit("greet.customgauge", 1.0);

```

重新启动服务器并转到/metrics 以查看已在其中反映的新规和计数器。

记录微服务

API 文档的传统方法是编写服务规范文档或使用静态服务注册表。当我们同时使用大量微服务的时候，很难保持 API 的文档同步。

微服务可以多种方式记录。本节将探讨如何使用流行的 Swagger 框架记录微服务。以下示例将使用 Springfox 库来生成 REST API 文档。Springfox 是一组 Java 和 Spring 友好的库。

创建一个新的 Spring Starter 项目并在库选择窗口中选择 Web。将项目命名为 chapter2.swagger。

提示

本示例的完整代码请参考本书代码文件中的 chapter2.swagger 项目。

由于 Springfox 库不是 Spring 套件的一部分，因此需要编辑 pom.xml 并添加 Springfox Swagger 库依赖项。将以下依赖项添加到项目中：

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.3.1</version>
</dependency>
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.3.1</version>
</dependency>
```

创建类似于先前创建的 REST 服务，也添加 @EnableSwagger2 注释，如下所示：

```
@SpringBootApplication
@EnableSwagger2
public class Application {
```

这是一个基本的 Swagger 文档所需要的。启动应用程序并将浏览器指向 <http://localhost:8080/swagger-ui.html>，会打开 Swagger API 文档页面。

如图 2-30 所示，Swagger 列出了 Greet Controller 的可能操作。单击“GET”，会显示当前项目中的 HTTP GET 请求。

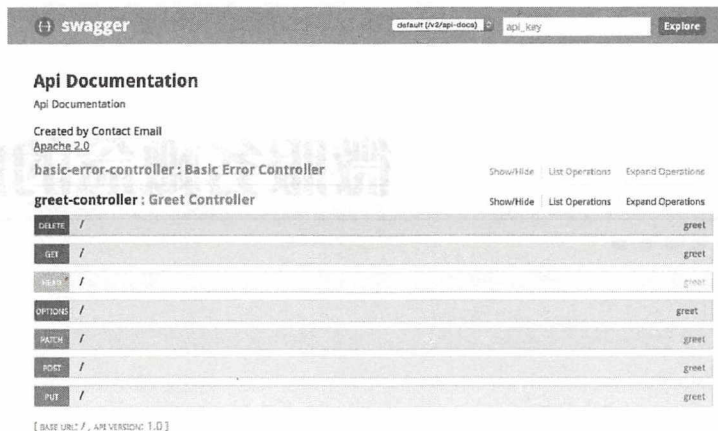


图 2-30

总结

在本章中，了解了 Spring Boot 及其关键功能，以构建生产级别的应用程序。

我们探索了上一代 Web 应用程序，然后展示了 Spring Boot 为开发微服务带来的便利。我们还讨论了服务之间基于异步消息的交互。此外，我们探讨了如何实现微服务所需的一些关键功能，如安全性、HATEOAS、跨源、配置等。我们还介绍了 Spring Boot Actuator 如何帮助运营团队，以及我们如何根据具体需求进行定制。最后，探讨了微服务 API 的文档化。

在第 3 章中，我们将更深入地研究在实现微服务时可能出现的一些实际问题，除此之外我们还将讨论一个功能模型。

第 3 章

微服务概念的应用



虽然微服务十分优秀，但是如果没有很好的设计，错误的理解了微服务，可能导致无法挽回的后果。

本章我们将会讨论在微服务的实际应用中所存在的技术难点，同时关于开发决定微服务的关键性决策我们也会给出指导性的建议。还会给出解决方案和模式来处理微服务的一些常见问题。本章还会回顾在企业级微服务开发中所存在的难题，以及如何解决这些难题。更为重要的是，在最后还会给出一个微服务生态系统的功能模型。

通过本章的学习，您将会学到如下知识：

- 开发微服务时需要考虑的不同设计方法和模式。
- 开发企业级微服务时的挑战。
- 微服务生态系统的能力模型。

模式和常见设计决策

微服务在近几年取得了巨大的成功，逐渐成为了架构师的首选，SOA 则退居其次。但同时也要明白一个事实，微服务是开发可扩展的原生云系统的利器，但成功的微服务需要细致的设计以避免产生灾难性的后果。微服务也绝非“万金油”，适用于所有的架构场景。

一般来说，对于开发一个轻量级的、模块化的、可扩展的分布式系统而言，微服务会是一个很好的选择。过度工程化和错误使用案例，很可能将系统变成灾难。想要开发一个成功的微服务，选择正确的用例是至关重要的，通过进行适当的权衡分析来采取正确的设计决策也是非常重要的。在设计微服务时，还有很多其他因素需要考虑，我们将会在下面做详细说明。

确定合适的微服务边界

与微服务相关的一个尤为常见的问题是考虑服务的大小。一个微服务究竟能够有多大（微型—单体应用），或者他能够有多小（纳米服务）？是否存在大小合适的服务？大小真的重要吗？这个问题的快速回答可以是“一个 REST 规范的接口对应一个微服务”，或者“微服务应该少于 300 行代码”，抑或“一个微服务组件只负责单一业务”。但是在我们选择三者中的任何一个答案之前，还是有很多东西可以分析，以便我们更好地理解服务边界的概念。

领域驱动设计（DDD）定义了有界上下文（bounded context）概念。一个有界上下文是一个负责执行特定功能的大的领域，或者系统的子领域或子系统。

提示

访问 <http://domainlanguage.com/ddd/> 以便了解更多关于 DDD 的内容。

图 3-1 是一个关于领域模型的例子：

对于一个金融财务后台，系统发票、会计、计费等代表不同的有界上下文。这些有界上下文是强度隔离的域，与业务能力密切相关。在经济领域，发票、会计及

账单分别属于不同的业务，它们通常由财务部门的不同下级组织所管控。

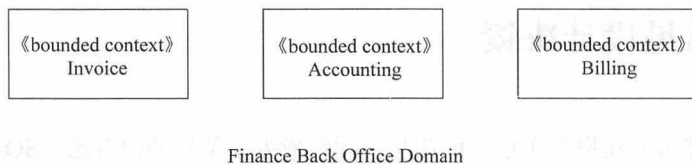


图 3-1

一个有界上下文是确定微服务边界的有效方式。每个有界上下文能够映射到单个微服务。在现实生活中，有界上下文之间的通信通常很少会成对出现，一般是不相关的。

尽管现实世界中有组织的边界是创建一个有界上下文最简单的机制，但是在某些场景下，由于在组织结构内部所遗留的一些问题，还是有可能带来错误的结果。举例来说，一种业务可能会通过前台、线上、漫游代理等多种渠道进行传递。在很多组织中，业务单元可能基于传递渠道而非具体的底层业务来进行组织。在某些场景下，组织边界可能并不能指明准确的服务边界。

一个由上至下的分解能够成为另一个创建正确有界上下文的方式。

没有固定的捷径可以用来确定微服务的边界，通常来说，确定边界这件事是非常具有挑战性的。在从单体应用到微服务整合的场景下，确定边界是相对容易的，因为服务边界和依赖在已有系统中是已经明确了。然而对于一个全新的微服务开发而言，在前期是很难确定相关依赖的。

设计微服务边界最实用的方法是通过一系列可能的选项来把手边的工作运行起来，就像是一个服务的“试金石”。记住，可能会有多种情况符合一个给定的场景，这时就需要进行对比权衡。

接下来的场景有助于定义微服务边界。

自治功能

如果我们审查的功能天生就是自治的，那么它就可以用来作为一个微服务边界。自治服务通常很少依赖外部功能，它们接收输入，通过内部的逻辑和数据进行计算，然后产生结果进行输出。所有公用功能，如加密引擎或者通知引擎就是很直观的例子。

再如，一个配送服务，接收一个订单，然后处理订单，最后通知配送人员进行配送。再如一个在线的航班搜索服务，它的查询结果基于缓存的可用座位信息，也是一个自治功能的例子。

可部署单元的大小

大多数微服务生态系统都会利用自动化，如自动集成、交付、部署及伸缩。微服务覆盖了更为广泛的功能，这也导致部署单元比较大。大的部署单元给自动复制文件、下载文件、部署和启动的时间带来了挑战。服务的大小随着它实现的功能密度的增加而增加。

好的微服务应该确保它可部署的单元的大小始终保持在可控范围内。

最合适的功能或者子域

分析如何从一个整体应用上分离出一个最有用的组件是非常重要的，这个特别适用于将整块的应用切分成一个个微服务。这种分析可以基于资源密集程度、所有权成本、商业利益以及灵活性等因素。

例如，在一个酒店预订系统中，50%~60%的请求都是与搜索相关的，把搜索功能移出去可以提高系统灵活性、增加商业利益、减少开支、释放资源。

多语言结构

微服务一个很重要的特点就是它对多种语言架构的支持。为了满足不同的非功能性和功能性的需求，不同组件可能需要区别对待。它们可能采用不同的架构，不同的技术，不同的部署和拓扑等。当组件一旦确定，需要考虑对语言的要求。

在前面提到的酒店预订的场景中，一个预订微服务可能需要事务完整性，但是搜索微服务则不一定需要。酒店预订微服务可能使用类似于 MySQL 的这种符合 ACID 原则的数据库，但是搜索微服务则可能会使用符合最终一致性的数据库，如 Cassandra。

选择性缩放

选择性缩放与前面提到的多语言结构是相关联的，与前面提到的例子类似，并

非所有的功能性模块都需要同样等级的可扩展性。有时候，根据可扩展性需求来决定边界是比较合适的。

举例来说，在酒店预订例子中，由于搜索请求比预订、通知等请求出现的频率要高得多，因此，相对于其他类型的微服务而言，需要更多地考虑搜索微服务的扩展性。因此，为了更好地响应速度，可以选择将搜索微服务独立出来，将其运行在弹性搜索的顶层或者将其数据保存在内存中。

小而灵活的团队

微服务允许小且集中的团队分别开发系统的不同模块。系统的各个部分分别由各个不同的组织进行创建，甚至跨区域合作或者团队成员技能参差不齐的场景，在制造业中尤其常见。

在微服务的世界中，每个小组分别开发不同的微服务，最后再把它们整合在一起。尽管这并不是一个划分系统最好的方式，还是可以借鉴的。

在一个在线商品搜索的场景，服务可能基于用户搜索的内容，为顾客提供个性化选择。这个可能需要复杂的机器学习算法，因此需要一个专业的团队。这个功能可以作为一个微服务，由不同的专业团队来共同进行实现。

单一职责

理论上讲，单一职责原则可以运用于一个方法、一个类或者一个服务上。但是，在微服务这一背景下，单一职责并非对应于单个服务或者某个端点。

一个更加合适的方式是把单一职责理解成单个业务能力或者单个技术能力。按照单一职责的原则，任何单个职责都不能由多个微服务共同承担，同样地，单个微服务也不应该同时承担多个职责。

不过，还是存在单个业务项被切分到不同的服务这种特殊情况。例如，管理客户资料时，为了达到一个好的工作效率，您可能采取命令查询责任隔离（CQRS）的方法来使用两个不同的微服务去管理读和写。

可复制性或变性

在 IT 交付中创新和速度是最为重要的。我们在定义微服务的边界时，需要使得

每个微服务都能很顺畅地从整个应用中分离出来，而不用再次去重复编写它的代码。如果系统的某个部分符合这种特征，那么它就很适合去作为一个微服务。

很多组织都认为最重要的事情是实现功能及快速交付。这些组织可能并不会太在意架构和技术本身。更多时候，他们更加关心什么样的工具或者技术能够最快地达到目标。组织越来越多地选择通过组合几个服务来开发最小可行产品（MVPs）的方法。系统发展往往伴随着代码重构和服务替换，这时微服务就起了关键性的作用。

耦合和内聚

耦合和内聚是决定服务边界最重要的两个因素。必须很认真地评估微服务之间的依赖以避免高耦合的接口。将功能进行依赖分解有助于确定微服务边界。避免服务间的频繁通信、过多的同步请求—响应调用、同步循环和依赖是 3 个很重要的关键点，这 3 点如果处理不好，会给服务之后的发展带来不可估量的危害。在微服务内部保持高内聚，微服务之间保持低耦合是一种比较成功的设计方式。除此之外，确保微服务之间的事务处理不会相互影响。最上层的微服务通过接收一个事件作为输入，执行一系列内部函数，最终发出另一个事件。整个计算过程中，它在自己的本地存储进行数据的读写。

把微服务看作一个产品

DDD 同样推荐把有界上下文看作一个产品。对于单个 DDD，每个有界上下文都是产品的理想选择。把微服务本身就看作一个产品。一旦微服务边界确定，就可从产品的角度去评估它们是否真的能够称为一个产品。业务用户相对来说更容易从产品的角度来看待微服务边界。一个产品边界可能有很多参数，如目标受众、部署的灵活性、销售能力、可重用性等。

设计通信方式

微服务之间的通信方式可以设计成为同步（请求—响应）或者异步（即发即弃）的通信方式。

同步通信方式

图 3-2 展示了一个使用请求/响应方式进行通信的例子。

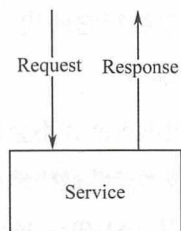


图 3-2

在同步通信中，不存在共享状态或者对象。当一个调用者请求一个服务，它发送请求的数据，然后等待返回结果。这种方式有很多优势。应用是无状态的，为了达到高可用的目的，可以运行多个活跃的实例，用以负载。由于没有其他类似共享消息服务这种基础依赖，需要的管理开销也就比较少。为了防止在任何环节出现异常，异常信息会立即反馈给调用者，保证系统的一致性，而不会损害到数据的完整性。

同步的请求—响应通信方式的缺点在于在请求被完全处理之前，用户或者调用方将会保持阻塞。后果就是，调用方的线程会阻塞地等待响应，因此，这种方式会限制系统的可伸缩性。

同步通信方式在微服务之间增加硬依赖，如果服务链中的一个服务不可用，将会导致整个服务链不可用。为了保证服务成功，每个依赖的服务都必须处于运行的状态。很多失败的场景需要使用超时和循环进行处理。

异步通信方式

图 3-3 展示了一个接收异步消息作为输入，然后异步返回结果供其他服务调用的服务。

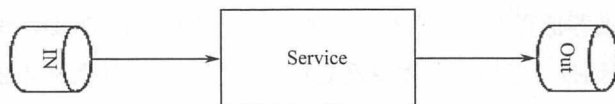


图 3-3

异步方式基于无回路语义，它对微服务之间进行了解耦。这种方式提供了高层次的可扩展性，因为服务是独立的，并且在负载增加时能够在内部产生线程进行处理。当发生超负荷时，消息将会进入到消息服务的队列中以待后续处理。这意味着即使某一个服务处理事件过慢，也不会影响到整个服务链。这在服务之间提供了高层次的解耦，因此维护和测试都会变得更为简单。

这种方式的缺点在于，它对外部的消息服务产生了依赖。处理消息服务的容错问题也很复杂。消息通常使用主动/被动语义。因此，处理连续可用性的消息系统更

难实现。由于通常会把消息进行保存，对于 I/O 处理能力也会有较高的要求。

如何进行抉择

两种通信方式都各有长短，仅使用一种通信方式开发一个系统显然是不可能的，需要根据用例将两种通信方式配合使用。原则上讲，异步方式适合于创建可扩展性强的微服务系统。但是，如果试图把一切都设计为异步的，将会导致系统的设计过于复杂。

图 3-4 展示了一个用户通过单击 UI 获取简介细节的例子，应该使用哪种通信方式进行设计？

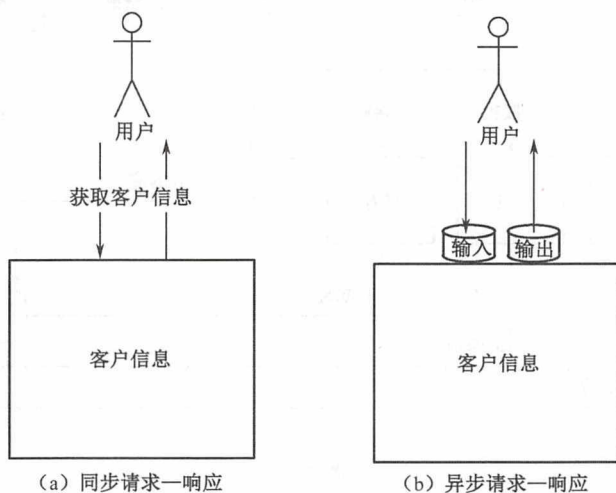


图 3-4

上面描述的是在请求-响应模型中，向后台系统发送请求，然后获取结果的一个简单的例子。同样可以使用异步通信方式进行建模，通过把消息入队至输入队列，然后在输出队列中等待对应 ID 的返回结果。虽然我们使用的是异步消息，用户在整个请求的过程中还是处于阻塞状态。

另一个用例是用户单击 UI 搜索酒店，如图 3-5 所示。

这看似和前面的例子很像，我们假设这项业务功能在返回酒店列表给用户之前，需在内部触发一系列活动。例如，当系统接收了这个请求，它计算了用户的评分、

根据目的地进行筛选、根据用户喜好进行推荐、根据客户价值和收入因素对价格进行优化等。在这个例子中，我们可以对这些因素中的一部分并行处理，然后再将结果汇总给用户。正如图 3-5 显示的那样，几乎所有的计算逻辑都可以插入到监听输入队列的搜索管道中。

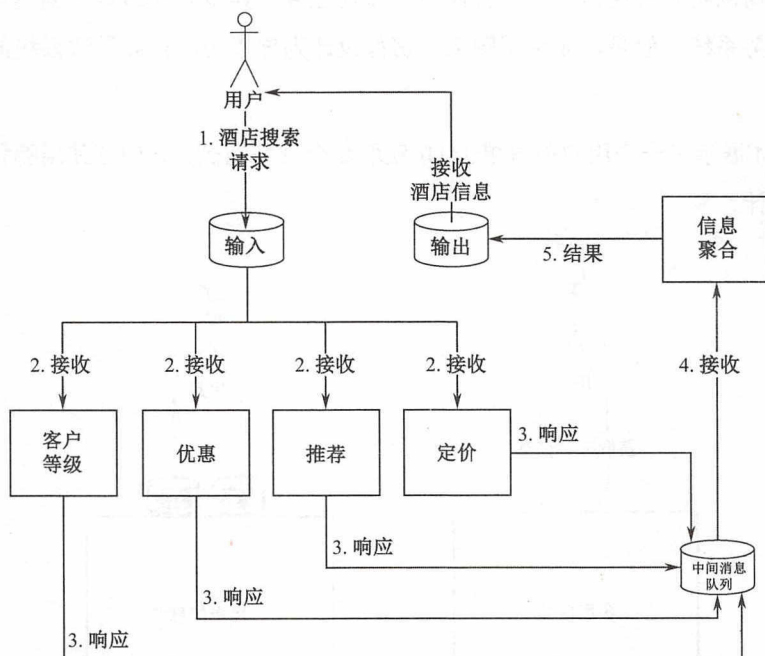


图 3-5

在这个例子中，建议首先使用同步请求响应的方式，如果后面发现有必要，再把它重构为异步方式。

下面的例子展示了一个完整的服务之间异步进行交互的方式，如图 3-6 所示。

这个服务在用户单击预订功能时进行触发。同样地，这是一个异步通信方式。当预订成功后，它会发送一条信息到用户的邮箱，发送一条消息到酒店的预订系统，更新缓存的库存信息和积分系统，准备发票，以及可能存在的其他操作。为了防止用户陷入长时间等待，应该对服务进行切分。只让用户等待到预订系统创建一条预订记录为止，一旦处理成功，一个预订时间就会发出，然后把确认消息回传给用户。随后，其他任务异步、并发进行处理。

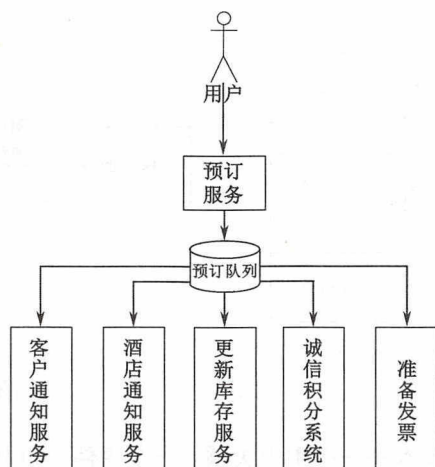


图 3-6

在上面的 3 个例子中，用户都需要等待服务的响应。通过新的 Web 应用框架，可以异步发送请求，然后定义回调函数或者给获取响应设置一个观察者。因此，用户并不会因为阻塞在这里而不能执行其他活动。

一般来说，在微服务领域，异步方式往往是更优的选择，具体选择哪种模式应该基于当前模式对系统带来的优点。如果使用异步方式进行建模没有任何优点，那么就使用同步方式，直到您发现一个可以令您信服的理由。使用异步通信方式对用户驱动请求进行建模会带来很多复杂性，使用反应式编程框架可以避免这一点。

微服务的编制

可组合性是服务设计原则之一。这导致谁对组合服务负责变得混乱。在 SOA 世界中，ESB 负责组合一组细粒度的服务，在一些组织中，ESB 作为代理的角色，服务提供者自己组成并暴露粗粒度服务。在 SOA 领域，针对这种情况有两种处理方法。

第一种方法是编制，如图 3-7 所示。

在编制方法中，多个服务组合在一起，共同实现一个完整的功能。一个中央大脑作为编制者。上图中订单服务作为一个复合服务对其他服务进行编制。在主进程中可能会分出串行或者并行分支，每项任务将由原子任务服务完成。在 SOA 中，ESB 担任编制的角色。编制服务由 ESB 作为混合服务暴露出来。

第二种方法是编排，如图 3-8 所示。

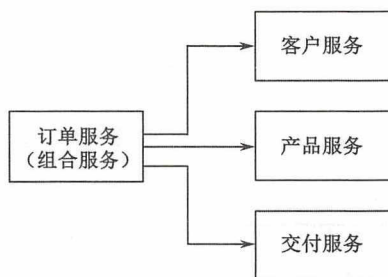


图 3-7

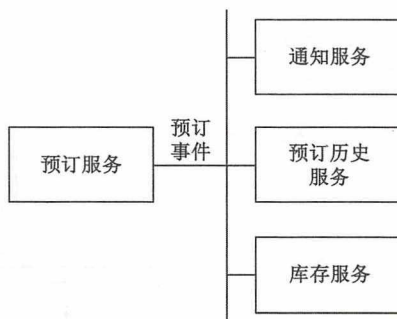


图 3-8

在编排方式中，并不存在一个中央大脑。一个事件（如上面的预订事件），由一个生产者发布，一系列消费者等待这个事件，并且在事件到来后互相独立地实施不同的处理逻辑。有时事件是嵌套的，消费者能够发送其他的事件供其他服务进行消费。在 SOA 中，请求方推送消息到 ESB，下游流向由消费服务方自动决定。

微服务是自治的，这意味着在理想状态下，完成功能所需的所有组件都应该包含在服务内部，包括数据库、内部服务的业务流程、内部状态的管理等，服务端提供粗粒度的 API。在不需要外部接触点的情况下，呈现完整的功能服务。但在现实中，微服务需要跟其他服务协作，共同完成功能。在这种情况下，编排是把多个服务连接在一起的更佳选择。依据自治原则，不建议使用脱离服务之外的组件控制处理流。如果用例能够用编排风格建模，那它可能会是最优的处理方式。

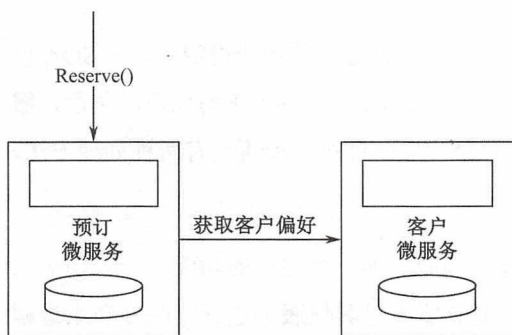


图 3-9

但编排风格并不适用于所有场景，图 3-9 描绘了这一点。

在前面的例子中，预订和用户是两个微服务，具有明确的职责划分。开发负责系统时，很可能会出现预订时获取用户偏好的情况。

那我们能否把用户服务移到预订服务之中，以便预订服务完全可以在内部处理完成？如果用户服务和预订服务有充分理由被划分成两

个微服务，就不应该再融合成一个。在这种情况下，我们稍后会介绍另一种单体应用。

我们能异步处理用户的请求和预订吗？

如图 3-10 所示，处理预订时需要获取用户偏好，因此，可能需要向用户服务发送一个同步阻塞的请求，此时不建议选择异步方式。

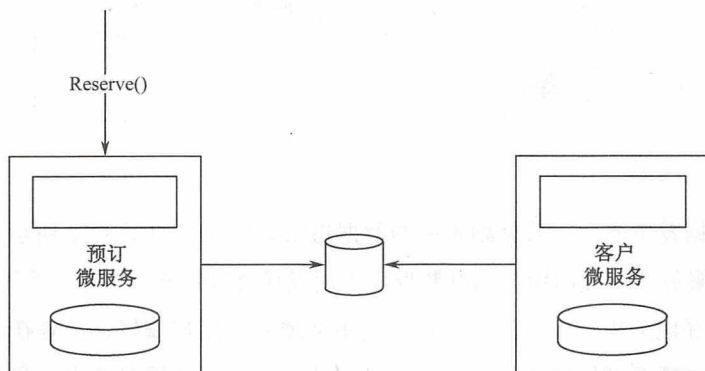


图 3-10

我们能否把业务流程单独抽离出来，然后创建另一个混合的微服务，通过它来组合预订服务和用户服务？

如图 3-11 所示，在服务内部组合多个组件是可行的，但是创建一个混合的微服务可能并非一个好的选择。我们最终将会在没有业务调整的前提下创建多个细粒度的非自治的微服务。

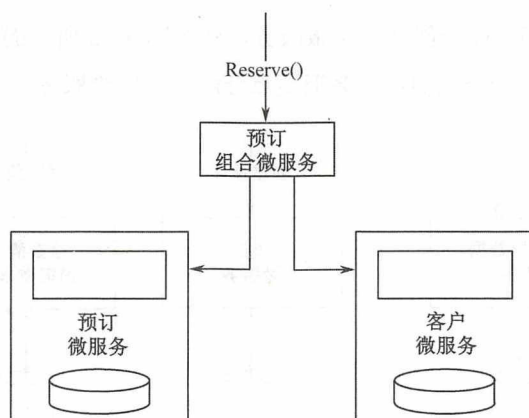


图 3-11

可以通过在预订服务中保留偏好数据的从属副本来复制客户偏好吗？

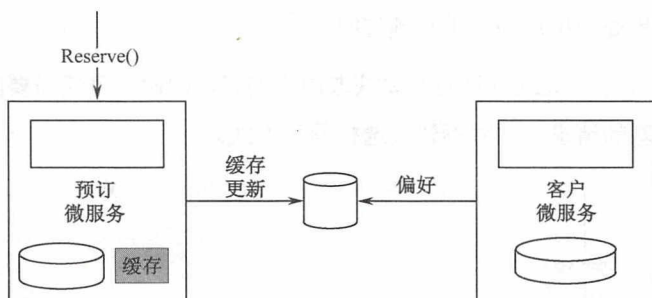


图 3-12

当主数据发生变化，数据副本中的数据也跟着更新。如图 3-12 所示，在这个例子中，预订服务可以使用用户偏好数据而不用发送外部请求。这是一个不错的想法，但是需要好好地分析一下。当下我们只是单纯地复制用户偏好，但是在其他的场景中，我们可能需要调用用户服务来查看该用户是否在预订黑名单中。我们需要很小心地决定什么数据该复制，这会增加问题的复杂度。

在微服务中有多少端点？

在很多情况下，开发者对于每个微服务中存在多少个端点感到困惑。这个问题本质上是在问：应该把每个微服务限制在一个端点还是多个端点？

端点的个数其实并非决策点。一个或多个端点的场景都可能出现。例如，考虑一个传感器数据服务负责收集传感器信息，有两个逻辑端点：创建和读取。但是为了处理 CQRS，我们可能会创建两个微服务，就像图 3-13 所示的酒店预订例子中的那样。语言结构是另一个我们可能会把端点分散到不同微服务中的场景。

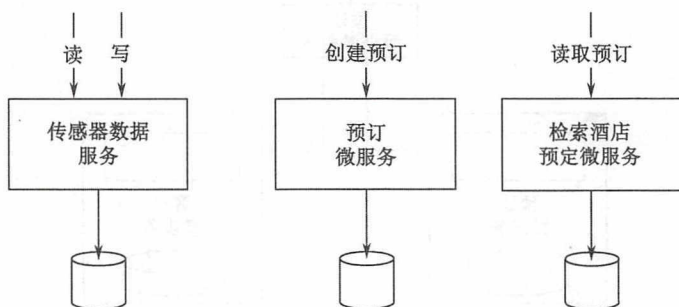


图 3-13

对于通知引擎，通知作为对一个事件的响应发送出去。准备数据、鉴定用户、发送机制等通知的过程都属于不同的事件。更进一步，在不同的时间窗口，我们可能希望以不同的方式来对每个过程的比例进行调整。针对这个场景，我们可以把每个通知端点拆分成一个个独立的微服务。

再举一个例子，一个信用积分服务可能会包含多个微服务。例如，积累、兑换、转移和结算，我们可能不想针对每个服务进行不同的处理。所有这些服务都是相连的并且使用同一个积分数据表。如果我们为每个服务设置一个端点，我们最终将会陷入很多细粒度的服务从同一个数据集中获取数据，或者同一个数据集存在多个副本这样的境地。

简而言之，端点个数不是一个设计决策，一个微服务可能有一个或多个端点。相较而言，为微服务设置合适的有界上下文更为重要。

一台虚拟机部署一个还是多个微服务？

一个微服务可以部署在多个虚拟机上，通过重复部署实现可扩展性和可用性，这点是毋庸置疑的。需要考虑的是，不同的微服务能否部署在同一个虚拟机上。这种方式有利有弊，当服务较简单并且流量较小时，就需要考虑这个问题。

考虑这样一个场景，我们有多个微服务，并且每分钟的总业务量少于 10，同时假设虚拟机的可用大小至少为双核 8 GB RAM。一个双核 8 GB RAM 的虚拟机每分钟可以处理 10~15 个业务而不用考虑性能问题。如果我们为每个微服务使用不同的虚拟机，成本会比较高，最终会支付很多费用在基础服务和证书上，因为很多厂商根据核心数量来收取费用。

处理这个问题最简单的方法是问以下几个问题：

(1) 虚拟机在高峰期是否有足够的容量同时运行这些服务？

(2) 我们是否想要区别对待这些服务以实现 SLAs（选择性缩放）？例如，为了可扩展性，如果我们有一个一体化虚拟机，我们复制虚拟机的同时也就复制了所有的服务。

(3) 有没有存在冲突的需求？例如，不同的操作系统版本，JDK 版本等。

如果对于上述问题，您的所有答案都是否，也许我们就可以开始配置部署，直

到遇到一个场景来改变部署拓扑。但是，我们需要保证这些服务并没有共享任何资源，并且是作为独立的系统进程来运行的。

前面已经提到，对于拥有成熟的虚拟基础设施和云基础设施的机构来说，这个可能并非什么大问题。开发者并不需要关心服务是在哪里运行的，开发人员甚至不用考虑容量问题，服务将会部署在一个计算云上。基于基础设施的可用性，SLAs 和服务性质，自我管理的基础设施部署，如 AWS Lambda。

规则引擎——共享还是嵌入？

对于任何系统而言，规则都是很重要的一部分。举例来说，offer 发放服务在做出是否发放 offer 之前会执行一系列规则。我们要么手工编码规则，要么使用规则引擎。很多企业在规则库中集中管理规则并且集中执行它们。这些企业的规则引擎主要用来为业务提供一个编辑和管理规则的机会，同时从中央仓库中重用这些规则。著名的规则引擎有 Drools，商业领域中 IBM、FICO 及 Bosch 已经成为这方面的开拓者。这些规则引擎提高了生产力，使得能够重用规则、事实、词汇，并且使用 rete 算法实现了更快速的规则执行。

对于微服务而言，一个中央规则引擎意味着把请求从微服务中分散到中央规则引擎。这同样意味着服务逻辑现在存在于两个位置，一部分在服务内部，另一部分在服务之外。然而，对微服务而言，其目的就是减少外部依赖。

如果规则足够简单、数量少、只在服务边界之内使用，并且不暴露给业务用户进行规则编辑，那么手工编码业务规则就优于依赖一个企业规则引擎，如图 3-14 所示。

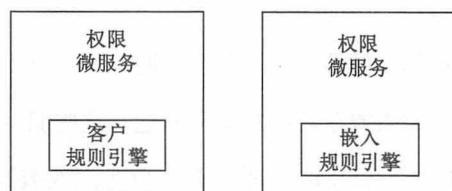


图 3-14

如果规则很复杂，限制在服务范畴，并且不提供给业务用户，那么在服务内部使用嵌入的规则引擎更好，如图 3-15 所示。

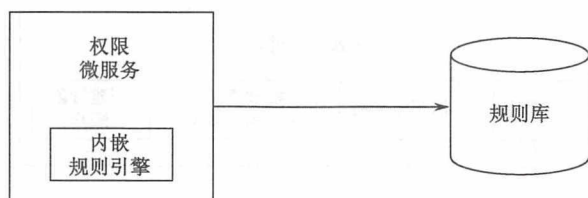


图 3-15

如果规则由业务所管理和委托，或者规则很复杂，抑或我们从其他服务域中重用规则，那么一个拥有本地嵌入执行引擎的中央代理库会是更好的选择。

需要注意的是，采用规则引擎需要很仔细地评估，因为所有的供应商可能都不支持本地执行的方式，并且也可能存在技术依赖，可以只在特定应用服务中运行规则等。

BPM 的角色和工作流程

业务流程管理（BPM）和智能业务流程管理（IBPM）是两个适用于设计、执行及监控业务流程的工具。

BPM 典型的用例是：

- 协调一个长时间运行的业务流程，其中一些流程由现有的资源实现，而另外一些流程，处于很小众的领域，并没有现成的实现。BPM 允许组合两种类型，并且提供一个端到端的自动化程序。这个通常涉及系统和人的交互。
- 以流程为中心的组织，如果已经实现了 6σ ，希望通过监控流程以达到效率的进一步提升。
- 通过重新定义组织的业务流程来彻底优化业务流程。

在两个场景下，BPM 很适合在微服务中使用。

第一个场景是业务流程重工程化，或者端到端的长时间运行的业务流程。如前所述，在这个场景中，BPM 操作在一个更高级别，它拼接一些粗粒度的微服务、已有的传统的连接器及与人的交互，从而实现功能交叉并且长时间运行的业务流程的自动化。正如图 3-16 中所展示的，贷款审批流程同时调用微服务和遗留应用程序服务，还集成了人的任务。

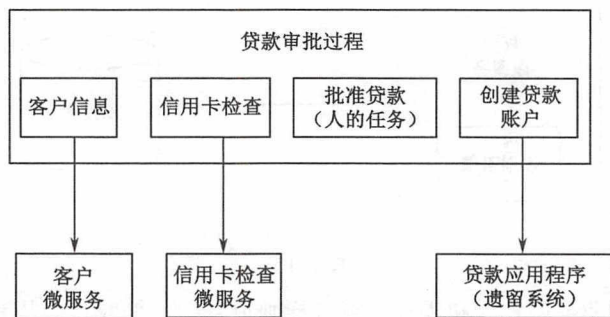


图 3-16

在下图场景中，微服务是一个实现了子流程的独立的服务。站在微服务的角度，BPM 只是另一个消费者。在这个方法中，需要注意避免从一个 BPM 过程中接受一个共享状态以及避免把业务逻辑转移到 BPM 上。

第二个场景是监控流程并且从效率上优化它们。与之同时出现的是一个完全自动化的、设计优秀的异步微服务生态系统。如图 3-17 所示，在这个例子中，微服务和 BPM 作为独立的生态系统进行运转。微服务在不同的时间点发送事件，如在流程开始时、状态发生变化时、流程结束时等。BPM 引擎利用这些事件来绘制和监控流程的状态。由于我们只是模拟一个业务流程来监控它的效率，因此可能并不需要一个成熟的 BPM 解决方案。订单交付并不是实施 BPM，更多的是一种监测仪表，捕捉和显示过程的进展。

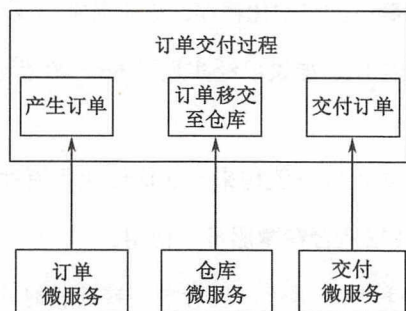


图 3-17

总的来说，BPM 仍然可以用在一个较高的层次，在由自动化系统与人交互建模成的端到端的跨职能的业务流程中，组合多个微服务。一个更好、更简单的方式是，

建立一个业务流程仪表盘（dashboard），微服务将第二个场景中提及的状态改变事件输入给它。

微服务能否共享数据存储？

原则上来说，微服务应该抽象化表现层、业务逻辑及数据存储。如果服务不符合上述规则，每个微服务逻辑上可以使用独立的数据库。

在图 3-18 中，产品和订单两个微服务共享同一个数据库和数据模型。共享数据模型、共享模式及共享数据表都会给微服务的开发带来灾难。这种方式一开始可能还好，但是当开发复杂的微服务时，我们就不得不在数据模型之间添加关联、添加查询等。这会导致物理数据模型紧密耦合。

如果一个服务只有几个表，它就没有必要使用像 oracle 数据库这样的完整数据库实例。在这种情况下，最好使用一个架构级的隔离，如图 3-19 所示。

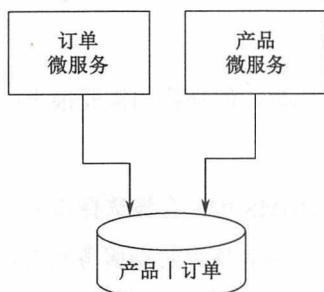


图 3-18

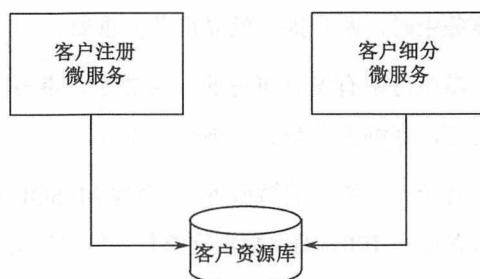


图 3-19

可能出现某些场景下，我们需要考虑在多个微服务间使用一个共享数据库。列举一个企业级管理的用户数据库或者主数据的例子，用户注册和用户区分微服务逻辑上共享同一个用户数据资源。

如图 3-20 所示，在这个场景下，一个可替代的方式是通过为这些服务添加一个本地业务数据存储来把微服务的业务数据存储从企业数据仓库中分出来。这会让微服务更加灵活地根据自身的需求去构造数据，并在本地数据库进行存储。当用户数据仓库发生任何改变时，企业用户仓库都会发送改变事件。同样地，如果在业务数据存储中发生了任何改变，这些改变也需要发送到中央用户仓库。

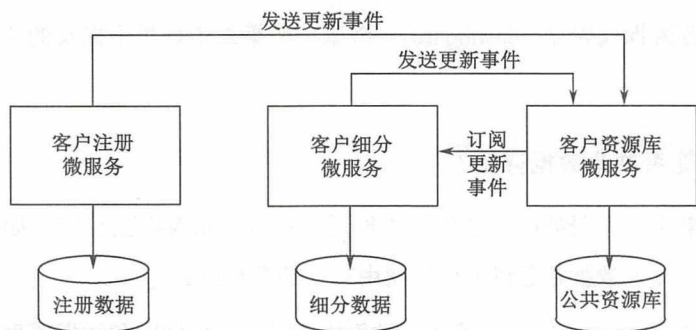


图 3-20

设置事务边界

事务在一个操作型系统中是通过把一系列操作组合成一个原子块来保证在一个 RDBMS 中存储的数据的一致性，它们要么提交要么回滚整个操作。分布式系统通过两步提交的方式来遵循分布式事务的概念，在异构组件如 RPC 服务、JMS 等参与到事务中时，两步提交就显得尤为重要。

微服务中有没有事务的一席之地？事务不是一件坏事，但是我们需要很小心地使用它，仔细斟酌是否必须用到事务。

对于一个给定的微服务，一个像 MySQL 这样的 RDBMS 可能会被选择作为一个后备存储来 100% 确保数据完整性。例如，对于一个库存或者库存管理服务而言，数据完整性是非常关键的。使用本地事务来定义微系统中的事务边界是很合适的。但是，在微服务中，应该避免分布式全局事务。为了尽可能确保事务边界不会跨越两个不同的微服务，适当的依赖分析是很有必要的。

改变用例以简化事务需求

最终一致性相对于跨越多个微服务的分布式事务而言，是一个更好的选择。最终一致性减少了很多开销，但是应用开发人员需要重新考虑他们编写应用代码的方式。这个可能包括重塑功能、顺序操作以减少失败、批量插入和修改操作、重塑数据结构，以及最后的补偿操作。

一个经典的例子是，在一个酒店预订用例中最后一个房间的销售场景。如果现在只剩最后一个房间，多个顾客同时预订这最后一个可用的房间时应当怎么处理？

一个业务模型的改变有时会使这个场景变得不那么复杂。我们可以设置一个“可预订房间列表”，预计到可能会有预订了然后取消的情况，实际可预订的房间数量可能低于实际可提供的房间数（可预订房间数=可提供房间数-3）。在此范围内的任何情况都会被接受为“有待确认”，用户只有在确认支付时才会被收取费用。预订将在一个设定的时间窗口确认。

现在考虑如下场景，我们使用类似 CouchDB 的 NoSQL 数据库创建客户档案。在更传统的使用 RDBMS 的方式下，我们首先插入一个客户，然后插入客户的地址、档案的详细信息、用户偏好，所有这些都放在一个事务中。当使用 NoSQL 时，我们可能不用做这些事。相反，我们准备好一个包含所有这些细节的 JSON 对象，然后一次性插入到 CouchDB 中。此时不需要显式的事务边界。

分布式事务场景

如果需要，理想的场景是在微服务中使用本地事务，并且完全避免分布式事务。可能出现这样的场景，在一个服务执行完以后，我们可能想要发送一条消息给另一个微服务。举个例子，假设一个旅游预约有轮椅的要求。一旦预约成功，为了预订轮椅，我们会发送一个消息给另一个处理辅助预订的微服务。这个预订请求本身会在一个本地事务中运行。如果这条消息发送失败，但我们还在这个事务边界中，则我们可以回滚整个事务。如果我们创建了一个预约并且发送了消息，但是发送消息之后，在预约过程中发生了错误以致预约失败，随后，预约记录发生回滚，这种情况该如何处理？我们现在陷入了一种不必要的单独预订轮椅的局面，如图 3-21 所示。

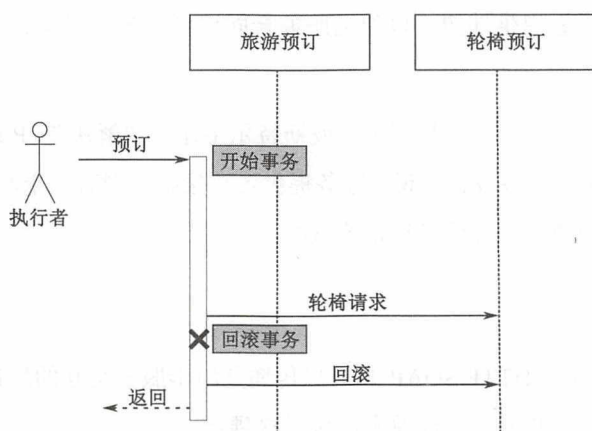


图 3-21

有两种方法可以解决这个问题。第一种方法是延迟发送消息直到预订完成，这个可以确保消息发送后失败的概率更小。第二种方式是，如果消息发送后发生了失败，触发异常处理机制，发送一个补偿消息来取消预订。

服务端点设计考虑

微服务中很重要的一点是服务设计。服务设计有两个关键的因素：协议设计与协议选择。

协议设计

服务设计的首要原则是简单。服务需要设计给用户去使用，一个复杂的服务协议会减少服务的使用率。保持简单和直接（Keep It Simple Stupid, KISS）原则有助我们更快地创建更高质量的服务，并且降低维护和替换的成本；YAGNI（You Ain't Gonna Need It）是另一个支持这个观点的原则。预测未来需求和创建系统是针对现在而不是面向未来的。这导致了大量的前期投资及更高的维护成本。

演化设计是一个很好的概念。只做足够的设计来满足当下的需求，根据需要不断改变和重构设计以适应新特性。之前说过，这个概念的实现需要一个强力的掌控者。

消费者驱动契约（CDC）是支持演化设计的一个很棒的观点。在很多情况下，当服务契约发生改变，所有使用了该服务的应用都要经受测试，这会给系统变更带来一定的难度。CDC 有助于在消费者应用上建立信心。CDC 提倡每个消费者以测试用例的形式向提供者提供期望，以便当服务合同被更改时，提供者使用它们进行集成测试。

波斯特尔定律也与这个场景相关，波斯特尔定律主要解决 TCP 通信，也同样适用于服务设计。对于服务设计来说，服务提供者在接收消费者请求时应尽可能灵活，而服务消费者应当遵守与提供者约定的规则。

协议选择

在 SOA 的世界，HTTP/SOAP 和消息传递是用于服务交互的默认服务协议。微服务遵循与服务交互相同的设计原则，包括松耦合。

微服务把应用分割成多个物理上独立部署的服务。这个不仅增加了通信成本，

也容易发生网络故障，还会导致服务性能低下。

面向消息的服务

如果我们选择异步通信方式，用户断开连接并不会导致响应时间受到直接影响。在这种情况下，我们可以使用标准的 JMS 或 AMQP 协议进行通信，用 JSON 作为负载。消息通过 HTTP 进行传递同样很受欢迎，因为它降低了复杂度。很多新生的消息服务都支持基于 HTTP 的通信。异步 REST 方式在调用长时间运行的服务时也很方便。

HTTP 和 REST 端点

使用 HTTP 通信总是具有更好的互操作能力、协议处理、流量路由、负载均衡、系统安全等。由于 HTTP 是无状态的，它更适合处理没有密切关系的无状态服务。大多数开发框架、测试工具、运行时容器、安全系统等都对 HTTP 友好。

随着 REST 和 JSON 的普及和接受，它们已经成为微服务开发者的默认选择。HTTP/REST/JSON 协议栈使得创建可交互操作系统变得非常容易和友好。HATEOAS 就是作为设计渐进渲染和自助导航而出现的一种设计模式。在第 2 章中曾提到，HATEOAS 提供了一种机制来把资源链接到一起，这样用户就可以在不同资源中进行切换。RFC 5988-WEB 是下一代标准。

优化通信协议

如果对服务响应时间的要求很严格，那么我们就需要特别关注与通信相关的问题。此时，我们可能需要在 Avro、Buffer 协议、Thrift 等多个协议之间来选择我们在服务之间进行通信的方式。但是这会限制服务之间的互操作性，我们需要在性能和互操作性需求之间进行权衡。自定义二进制协议需要进行仔细的评估，因为它们同时在消费方和服务方绑定了 native 对象。这个可能带来发布管理问题，如在基于 JAVA RPC 类型的通信中可能会出现类版本不匹配。

API 文档

最后一件事：一个好的 API 不单单是简单，还需要给调用方提供足够的文档说明。如今有很多工具可以用来编写基于 REST 风格服务的文档，如 Swagger、RAML

和 API Blueprint 等。

处理共享库

微服务背后的原则是它们要自治和独立。为了匹配这一原则，在某些情况下，我们不得不复制代码和库，包括技术库或者功能组件，如图 3-22 所示。

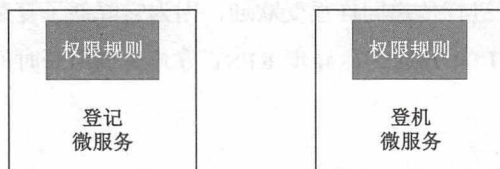


图 3-22

例如，飞行升级的资格将在办理登机手续及登机时审查。如果办理登机手续和登机是两个不同的微服务，我们可能需要在两个服务上重复编写资格审查规则。此时需要在添加依赖和代码重复上进行权衡。

与引入外部依赖相比，嵌入代码可能相对简单，因为它保证了更好的发布管理和应用的性能。但是这个违反了 DRY 原则。

提示

DRY 原则：在系统内的每一个知识点必须有一个单一的、明确的、权威的代表。

这种方式的一个缺点在于，如果在共享库上存在一个 bug 或者需要新增功能，就不得不在多个地方进行升级。这个可能并非一个很大的问题，因为每个服务可以包含一个不同版本的共享库，如图 3-23 所示。

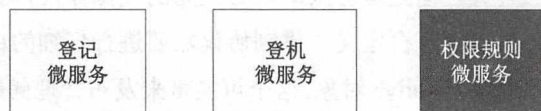


图 3-23

有一种解决方式是，把共享库本身开发成为另一个微服务。不过采取这种方式需要慎重。如果从业务能力角度看，它不能作为一个微服务，这带来的复杂性是大于实用性的。需要在多个服务间通信上的开销和库的重复之间进行权衡。

微服务中的用户接口

微服务的原则主张把一个微服务作为从数据库到表现层的垂直切片，如图 3-24 所示。

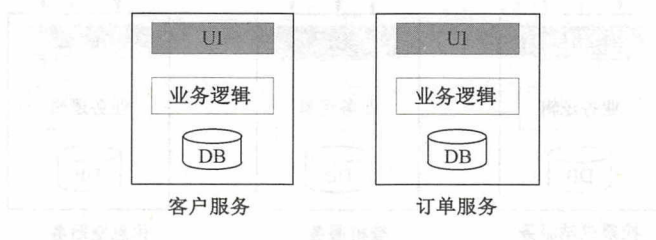


图 3-24

在现实中，我们经常需要配合已有的 API 建立快速的用户界面和移动应用。一个业务需要从 IT 中获取快速转变时间，如图 3-25 所示。

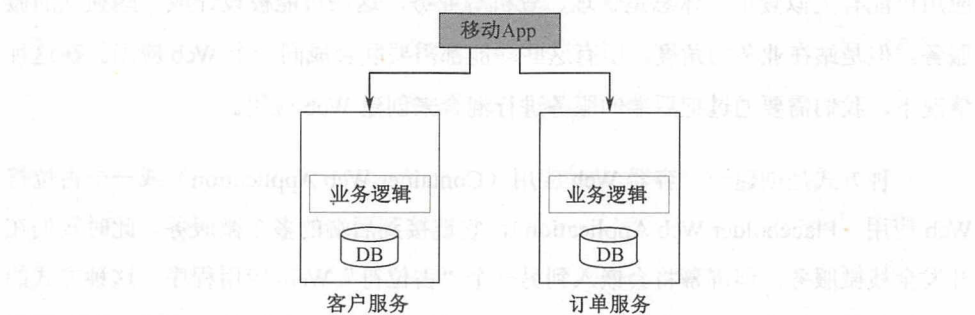


图 3-25

移动应用的进入是这种方式的诱因之一。在很多组织中，移动开发团队和业务团队紧坐在一起，通过连接和搭配内部及外部等多个源头的 API 来快速开发移动应用。在这种情况下，后台可能只是暴露服务，由移动开发团队自己来实现业务需求。此时，我们可以创建无图形界面的业务应用或服务，留给移动开发团队自己创建表现层。

另一类问题是，企业可能希望建立针对社区的综合 Web 应用程序，如图 3-26 所示。

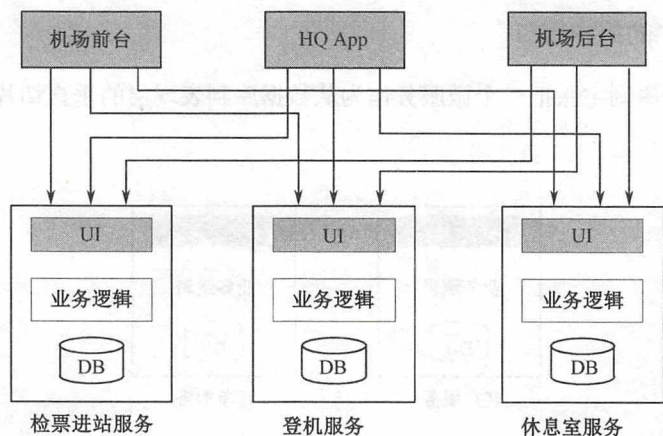


图 3-26

例如，企业可能希望开发一个针对机场用户的离境控制应用，一个离境控制 Web 应用可能有类似登记、休息室管理、登机等业务，这些可能被设计成一些独立的微服务。但是站在业务的角度，所有这些功能都需要联合成同一个 Web 应用。在这种情况下，我们需要通过把后端的服务进行混合来创建 Web 应用。

一种方式是创建一个容器 Web 应用（Container Web Application）或一个占位符 Web 应用（Placeholder Web Application），它链接到后端的多个微服务。此时我们在开发全栈微服务，但屏幕将会嵌入到另一个“占位符”Web 应用程序。这种方式的一个好处是，您可以有多个“占位符”Web 应用分别针对不同的用户群，像上图展示的那样。我们可以使用一个 API 网关来避免这些交错的连接，在下一节我们会探讨 API 网关。

在微服务中使用 API 网关

随着类似 AngularJS 等客户端 JavaScript 框架的发展，服务方都希望能暴露 RESTful 服务。这个会带来两个问题，第一个问题是协议期望的不匹配，第二个问题是访问一个页面时需要向服务器发送多个请求。

我们从协议期望不匹配这个问题开始，例如，GetCustomer 可能返回一个带有多个字段的 JSON 串。

```
Customer {  
    Name:  
    Address:  
    Contact:  
}
```

在前面的情况下，Name、Address 和 Contact 是嵌套的 JSON 对象。但是一个移动端用户可能只需要类似 first name 和 last name 这种基础的信息。在 SOA 的架构下，ESB 或者一个移动端中间件为客户端做了数据转换这项工作。在微服务中，默认的方式是获取 Customer 所有的元素，然后由客户端自己来自行过滤这些内容。在这种情况下，主要开销在网络上。

针对这个问题，我们可以有多种解决方式：

```
Customer {  
    Id: 1  
    Name: /customer/name/1  
    Address: /customer/address/1  
    Contact: /customer/contact/1  
}
```

在第一个方法中，我们只发送最少的信息，在其中附带其他内容的链接，这种方式我们在 HATEOAS 一节中曾经介绍过。在前面的例子中，对于用户 ID 1，有三个链接可以帮助用户访问特定的数据元素。这个例子只是一个简单的逻辑表示，不是一个真实的 JSON 串。移动客户端在这种情况下可以获取基本的用户信息，随后可以使用这些链接来获取额外所需要的信息。

当客户端发送 REST 请求时可以采取第二种方式。客户端在发送请求时，附带所需要的字段作为请求的一部分。在这种情况下，客户端发送一个请求，附带 firstname 和 lastname 作为查询字段，来表示客户端只查询返回这两个字段。这种方式的缺点在于，它会使服务端的逻辑变得复杂，因为服务端需要根据这些字段来过滤需要返回的数据，服务器必须根据到来的请求返回不同的元素。

第三种方式是引入一个间接层。在客户端和服务端之间添加一个网关组件，根据客户端的需求对数据进行转换。这是一个更好的方式，因为我们不必根据服务契约进行妥协。这个过程中引入了所谓的 UI 服务。在很多情况下，API 网关作为后端

的代理，暴露一系列与用户相关的 API。

我们有两种方式可以部署 API 网关，第一种是图 3-27（a）中展示的一个微服务搭配一个 API 网关，第二种方式是为多个微服务共用一个公共的 API 网关，如图 3-27（b）所示，选择哪种方式取决于我们的目的是什么。如果我们使用 API 网关作为反向代理，那么现成的，如 Apigee、Mashery 等，类似的网关可以作为一个共享平台。如果我们需要对流量整形和复杂的转换进行细粒度的控制，那么针对每个服务自定义网关可能更有用。

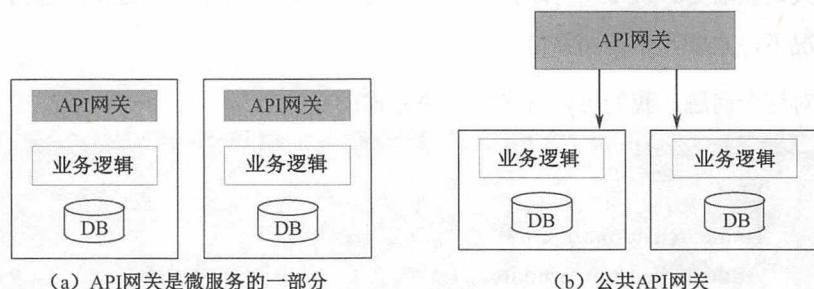


图 3-27

一个相关的问题是，我们将会从客户端发送很多请求到服务端。如果参考我们在第 1 章所列举的旅游门户的例子（*Demystifying Microservice*），为了渲染每个组件，我们都要向服务器发送请求。尽管我们只是传输数据，网络传输带来的消耗仍然非常巨大。这种方式并非完全错误，正如在很多情况下，我们采用响应式设计和渐进式设计。根据用户导航，在有需要时，数据才会加载。为了实现这一点，客户端的每个组件应该以懒惰模式向服务端发送独立的请求。如果带宽是一个需要考虑的问题，那么 API 网关可以解决这一点。一个 API 网关作为中间件在多个微服务之间组合和转换 API。

在微服务中使用 ESB 和 iPaaS

从理论上讲，SOA 并不全是关于 ESB，但在现实中，ESB 通常处于很多 SOA 实现的中心位置。那么在微服务的世界中，ESB 扮演了一个怎样的角色？

通常，微服务是更小的完全的原生云系统。微服务轻量级的特征有助于实现部署和拓展自动化。反过来，企业级 ESB 本质上是重量级的，大多商业化 ESB 对云的

支持都不友好。ESB 的关键特征是协议调解、转换、编排及应用适配器。在一个典型的微服务生态系统中，我们可能并不需要上述的任何特点。

那些在 ESB 中与微服务相关的有限功能，都可以由 API 网关这样更加轻量级的工具来实现。业务流程从中央总线中移到微服务本身。因此，在微服务内部，并不需要集中的业务流程能力。由于服务之间普遍使用 REST/JSON 进行通信，所以并不需要协议调解。我们从 ESB 中得到的最后一个功能是连接到遗留系统的适配器。在微服务中，服务本身提供了一个具体的实现，因此，并不需要遗留系统的适配器。基于所有这些原因，在微服务中并没有 ESB 的一席之地。

很多组织创建 ESB 作为他们应用集成的骨干（EAI）。这种组织中的企业架构策略是根据 ESB 建立的。可能有一些企业级的策略，如审计、日志、安全性、验证等，当集成 ESB 时就已经有现成的可供使用。但是微服务倡导一个更加分散的管理，如果 ESB 集成微服务将会是矫枉过正。

并非所有服务都是微服务。企业有遗留应用、供应商应用等，遗留服务使用 ESB 来连接微服务。在企业级集成遗留系统和供应商应用的场景上，ESB 还是可以发挥自己的作用。

随着云技术的发展，ESB 的能力不足以管理云和云之间、云到本地等的集成。集成平台即服务（iPaaS）被引入作为下一代应用集成平台，这个进一步减弱了 ESB 的角色。在典型的部署中，iPaaS 调用 API 网关访问微服务。

服务版本控制的考虑

如果我们允许服务发展，其中一个需要着重考虑的方面就是服务版本控制。服务版本需要在前期考虑好，而不是事后去弥补。版本控制有助于我们发布新服务而不影响当前用户的使用。老的版本和新的版本在两边同时部署。

语义版本被广泛应用于服务版本控制。一个语义版本有 3 个组成部分：主版本号、次版本号及补丁版本号。当有一个突破性改变时，需要更新主版本号，当有向后兼容改变发生时使用次版本号，当有向后兼容 bug 修复时使用补丁版本号。

当服务中不止一个微服务时，版本控制会变得复杂。与在操作层面上相比，在服务层面上管理服务的版本相对简单。如果在操作上面有一点改变，服务被升级和

部署为 Version 2，版本更改适用于所有操作。这是不可变服务的概念。

规定 REST 服务的版本有 3 种方式：

- URI 版本控制
- 媒体类型版本控制
- 自定义消息头

在 URI 版本控制中，版本号包含在 URL 本身。在这种情况下，我们只需要关心主版本。因此，如果有一个次版本或者补丁版本需要改变，服务消费方并不需要关心。把最新的版本命名为一个非版本的 URL 是一个不错的方式，如下所示：

```
/api/v3/customer/1234
/api/customer/1234 - aliased to v3.
@RestController("CustomerControllerV3")
@RequestMapping("api/v3/customer")
public class CustomerController {
}
```

一个稍微不同的方法是把版本号作为 URL 参数的一部分：

```
api/customer/100?v=1.5
```

在媒体类型版本控制中，版本由客户端在 HTTP Accept Header 中进行设置：

```
Accept: application/vnd.company.customer-v3+json
```

在自定义 header 中设置版本的方式则比较低效：

```
@RequestMapping(value =("/{id}", method = RequestMethod.GET, headers = {"version=3"}))
public Customer getCustomer(@PathVariable("id") long id) {
    //other code goes here.
}
```

URI 方案便于客户端消费服务，但这有一些固有问题，例如，事实上版本嵌套的 URI 资源可能很复杂。确实，集成客户端与媒体类型方案相比稍显复杂，存在服务的多个版本的缓存等问题。但是，瑕不掩瑜，URI 方案还是很适合的。大多数互联网行业的佼佼者，如 Google、Twitter、LinkedIn 及 Salesforce 都遵循 URI 方案。

跨域设计

使用微服务，对于服务会在同一个主机或者同一个域下运行没有任何保证。混

合 UI Web 应用在实现一个任务时可能访问多个微服务，并且这些服务可能来自不同的域和主机。CORS 允许浏览器客户端发送请求到存在于不同域下的服务，这在基于微服务的架构中是必须的。

一个方法是使所有微服务允许来自其他可信的域的跨域请求，第二个方法是使用一个 API 网关作为一个单个可信域提供给客户端。

处理共享的参考数据

当开发大型应用程序时，我们发现一个常见的问题是对主数据或者参考数据的管理。参考数据更像是共享数据，不同的微服务都需要。城市主数据、国家主数据等将会被用于多个服务中，如航班时间表、航班预订等。

有一些方法可以解决这个问题。例如，对于那些相对静态、从不改变的数据，每个服务可以把这些数据硬编码到服务内部。

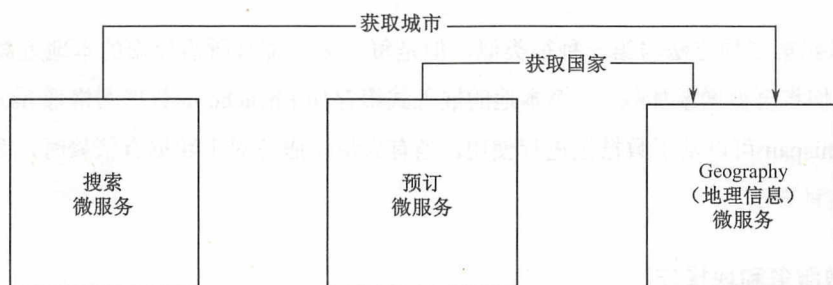


图 3-28

另一个方法，如图 3-28 展示的那样，是把它创建为另一个微服务。这样的编码是很简洁的，但是缺点是每个服务可能需要多次访问主数据。正如搜索和预订例子的图片中所展示的那样，Geography 微服务会访问共享数据，如图 3-29 所示。

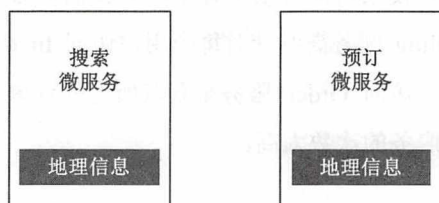


图 3-29

另一个选项是在每个微服务中复制一份数据，如图 3-30 所示。数据没有单一拥有者，每个服务都有他们需要的主数据。当有更新需要时，所有的服务都进行更新。这对性能是极度友好的，但是需要把代码在所有服务上复制一遍，这会给微服务之间同步数据带来不便。这种方式适合于基准代码和数据比较简单或者数据是比较静态的。

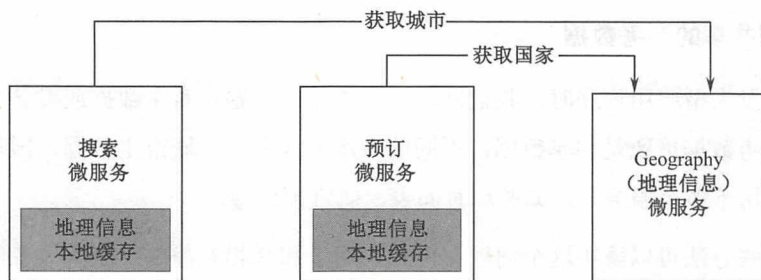


图 3-30

虽然第二种方法与第一种很类似，但是每个服务都有所需数据的本地近高速缓存，它们将会被增量加载。一个本地的嵌入式缓存如 Ehcache 或数据网格像 hazelcast 或 Infinispan 可以基于数据量进行使用。当有大量微服务对主数据有依赖时，很推荐采取这种方法。

微服务和块操作

由于我们已经把整块的应用切分成了更小的、目标明确的服务，也就不再可能在微服务数据存储之间使用 join 查询。此时可能导致一种情况，一个服务可能需要很多来自其他服务的记录来完成自己的功能。

例如，一个月度账单功能需要很多客户的发票来处理账单，如图 3-31 所示。我们使问题稍微复杂一点，发票可能有很多订单。当我们把账单、发票、订单分到 3 个不同的微服务中，Billing 服务需要针对每个用户访问 Invoices 服务来获取所有发票，然后对于每个发票，访问 Order 服务来获取所有的订单。这并非一个好的解决方案，因为访问其他微服务的次数太高。

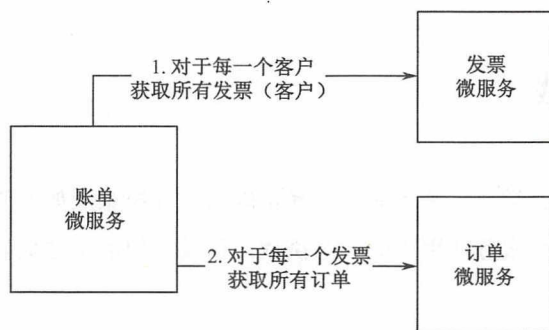


图 3-31

有两个解决方案。第一个方案是当数据创建时预聚合数据。如图 3-32 所示，当一个订单被创建时，一个事件被发出。当接收到这个事件，Billing 微服务保持内部每月汇总数据处理。在这种情况下，不需要 Billing 微服务访问外部服务来处理。这种方式的缺点是会有数据重复。

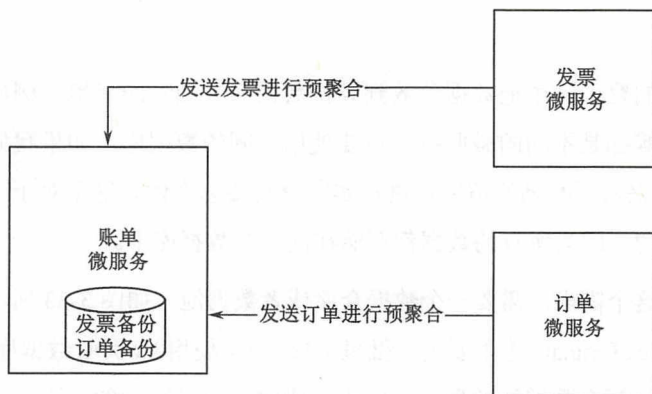


图 3-32

第二个方案是，当预聚合不太现实，就使用批量 API。在这种情况下，我们调用 `GetAllInvoices`，然后通过批量发送进一步使用并行线程来获取订单。Spring Batch 可以在这个场景下使用。

微服务的挑战

在上一节中，您学习了关于如何选择正确的设计决策及如何进行权衡。在这一节中，我们会回顾微服务使用中的一些挑战，以及如何解决他们以开发一个成功的微服务。

数据孤岛

微服务抽象它们自己的本地事务存储，用来达到它们自己的事务目的。存储的类型和数据结构将由微服务提供的服务进行优化。

举例来说，如果我们想要开发一个客户关系图，我们可能使用一个图形数据库，如 Neo4j、OrientDB 等。查找基于客户的相关信息，如护照号、地址、E-mail、电话等的预文本搜索，最好使用一个像 Elasticsearch 和 Solr 这样的索引搜索数据库来实现。

这将把我们置于一个把数据分散到异构数据岛的独特的境地。例如，客户、信用积分、预订等都是不同的微服务，需要使用不同的数据库。如果我们想通过结合这 3 个数据库来对所有高价值客户进行实时分析要怎么做？这个对于一个整块的应用来说是简单的，因为所有的数据都存储在同一个数据库中。

为了满足这个需求，需要一个数据仓库或者数据池，如图 3-33 所示。传统的数据仓库如 Oracle、Teradata 等主要用于批量上报。但是使用 NoSQL 数据库（如 Hadoop）及微批量技术，配合数据池的概念，实现近实时分析是可能的。不同于传统的为了实现批量上报所创建的数据仓库，数据池单纯存储数据而不去考虑这些数据将会被如何使用。现在的问题实际上是如何把数据从微服务中存储到数据池中。

把数据从微服务中存入一个数据池或者一个数据仓库中可以通过多种方式实现，如传统的 ETL。由于使用 ETL 时我们允许后门进入，并且打破了抽象，不能算是一个有效的转移数据的方式。一个当事件发生时，从微服务中把它们发送出去，如用户注册、用户更新事件等。数据摄入工具消费这些事件，然后适当地将状态改

变传播到数据池。数据摄取工具是高度可扩展的平台，如 Spring 云数据流、Kafka、Flume 等。

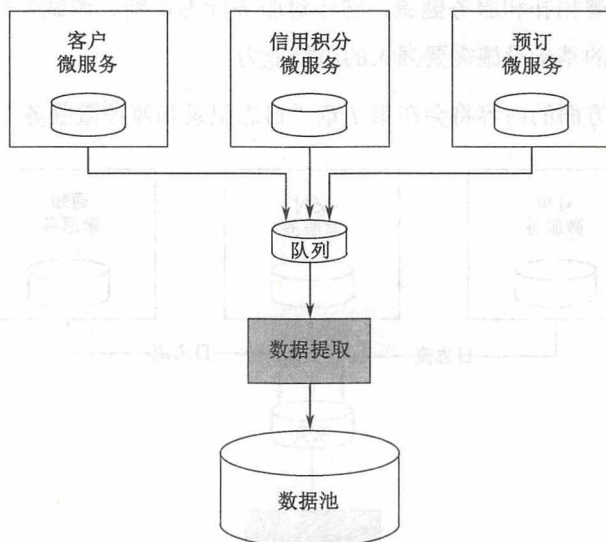


图 3-33

日志和监控

日志文件在分析和调试时能提供很有用的信息。由于每个微服务被独立部署，它们生产单独的日志，可能存到本地磁盘。这会导致碎片化的日志文件。当我们在多台机器上扩容服务时，每个服务实例可以产生各自的日志文件。这使得通过日志挖掘来调试和理解服务的行为非常困难。

检查订单、配送及通知作为 3 个不同的微服务，即便客户交易同时执行这 3 个微服务，我们也无法将它们和这个交易联系起来。

当实现微服务时，我们需要一个将日志从每个服务传输到中央托管日志存储库的功能，如图 3-34 所示。有了这个方法，服务就不再需要依赖本地硬盘或者本地 I/O。第二个好处是日志文件是由中央托管的，可供所有类型的分析，如历史的、实时的及趋势的。通过引入一个相关的 ID，端到端事务能够很轻易地进行跟踪。

由于微服务数量很大，并且有多个版本和多个实例，查看哪个服务是运行在哪

个服务器上、这些服务是否运行健康、服务依赖等都会变得复杂。这对于使用特定的或固定的服务器集群的单体应用程序来说要容易得多。

除了理解部署拓扑和服务健康，它还对服务行为识别、调试及热点识别带来了挑战。管理这样的基础设施需要强大的监控能力。

日志和监控方面的内容将会在第 7 章“日志记录和监控微服务”进行讲解。

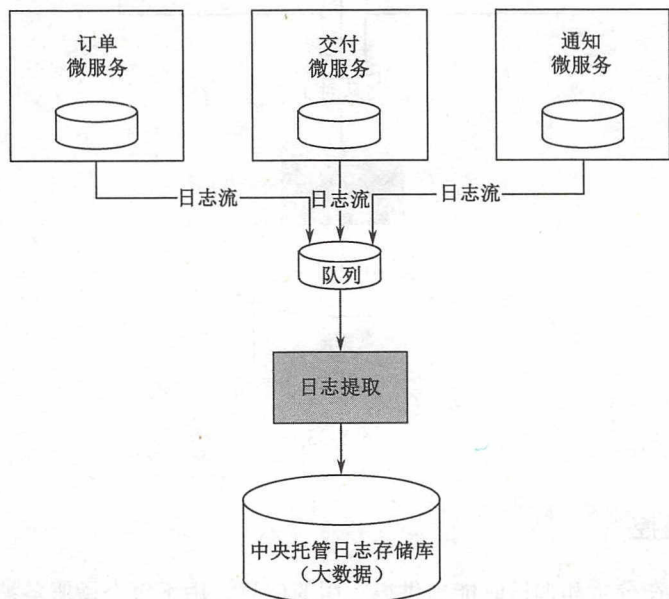


图 3-34

依赖管理

在大型微服务开发中，依赖管理是一个关键性的问题。我们如何鉴定及减少改变所带来的影响？我们如何得知是否所有的依赖服务都正在运行？如果其中一个依赖的服务不可用，服务应该如何应对？

太多的依赖会对微服务带来挑战。4 个重要的设计方面说明如下：

- 通过合理地设计服务边界减少依赖。
- 通过设计依赖时尽量低地耦合来减少影响。同样，通过异步通信风格设计服务交互。



- 使用类似断路器的设计模式来解决依赖问题。
- 使用可视化依赖图监控依赖。

组织文化

微服务实现中一个最大的挑战之一是组织文化。为了实现微服务交付的速度，组织应该采用敏捷开发过程、持续集成、自动化质量检测、自动化交付流程、自动化部署及自动化基础设施供应。

有着瀑布流开发模式或者重量级的发布管理过程与不频繁的发布周期这样的组织不利于进行微服务的开发。缺乏自动化也会给微服务部署带来挑战。

简而言之，云和 DevOps 都支持微服务开发的各个方面，这些都是实现成功的微服务所必需的。

管理面临的挑战

微服务实行分散治理，这和传统的 SOA 管理方式有很大不同。组织可能会发现很难适应这个改变，这对微服务的开发会带来负面的影响。

分散治理模式会带来很多的挑战。我们如何理解谁正在消费服务？我们如何保证服务重用？我们如何定义在组织中哪个服务是可用的？我们如何确保企业政策的执行？

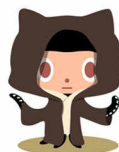
首要的事情是要有一套关于如何实现更好的服务的标准、最佳实践及指导说明。这些可以标准库、工具及技术的形式提供给组织。这确保了开发的服务是最高质量的，并且是以同一套方式进行开发的。

第二个需要考虑的是要有一个平台，所有利益相关者在这里不仅能够查看所有的服务，而且还能看到它们的文档、约定及服务级的协议。Swagger 和 API Blueprint 可以实现这样的功能。

操作费用

微服务开发通常增加可部署单元和虚拟机（或容器）的数量，这显著增加了管理成本和运营成本。

对于一个单一应用程序，在数据中心配置一定数量的专门的容器或者虚拟机可



能没有什么意义，除非业务利益非常大。成本通常随着规模经济而降低。把大量的微服务部署在一个完全自动化的共享基础设施中更有意义，因为这些微服务不与任何特定虚拟机或者容器所绑定。基础设施自动化、供应、集装箱式部署等功能对于大规模微服务的部署是必不可少的。如果没有这种自动化，操作开销和成本都会显著增加。

随着微服务数量的增多，可配置项目（CIs）的数量也变得很高，这些 CI 部署的服务器数量也可能变得不可预测。这使得用传统配置管理数据库（CMDB）管理数据变得极度复杂。在很多情况下，动态发现当前运行的拓扑比一个静态的配置 CMDB 类型的部署拓扑要更有用。

测试微服务

微服务同样给服务的测试带来了挑战。为了实现一个全方位的功能，一个服务可能依赖其他服务——以同步或者异步的方式。问题是我们如何测试一个端到端的服务，以评估其行为？而且依赖的服务在测试的时候可用状态是不确定的。

如果想要在测试服务时不依赖于其他服务，可以采用服务虚拟化或者服务模拟。在测试环境下，当服务不可用，模拟真实服务的行为。微服务生态系统需要服务虚拟化能力。但是，不能对其抱有太大的信心，因为模拟服务可能不能模拟很多边界情况，尤其是有深层次的依赖时。

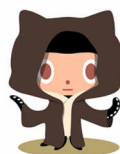
另一种方式，像我们之前提到的，是使用消费者驱动契约。转化的集成测试用例可以或多或少覆盖到所有服务调用的边界情况。

自动化测试、适当的性能测试和连续交付途径如 A/B 测试、功能发布控制、金丝雀测试、蓝绿部署、红黑部署，这些措施都降低了生产发布的风险。

基础设施配置

前面简要介绍了一下操作费用，人工部署会严重挑战微服务的推广。如果一个部署有人工元素，部署人员或操作人员应该了解运行拓扑、手动重新路由流量，然后一个个部署应用直到所有服务都升级完成。一旦服务实例增多，会显著增加操作费用。更进一步，这种人工方式的错误率也会增高。

微服务需要支持能够自动设置虚拟机或者容器、自动部署应用、调整流量、在



所有实例上复制新版本、优雅地淘汰旧版本的弹性云基础设施。自动化还要在有需要时通过增加容器或虚拟机来增加规模，并且在负荷低于阈值时缩小规模。

在有很多微服务的大型部署环境中，我们可能还需要额外的工具来管理能够自动初始化或销毁服务的虚拟机或者容器。

微服务能力模型

在对本章进行总结之前，根据本章提及的设计指南、常见的模式和不同情形的解决方案，我们来回顾一下微服务的能力模型，如图 3-35 所示。

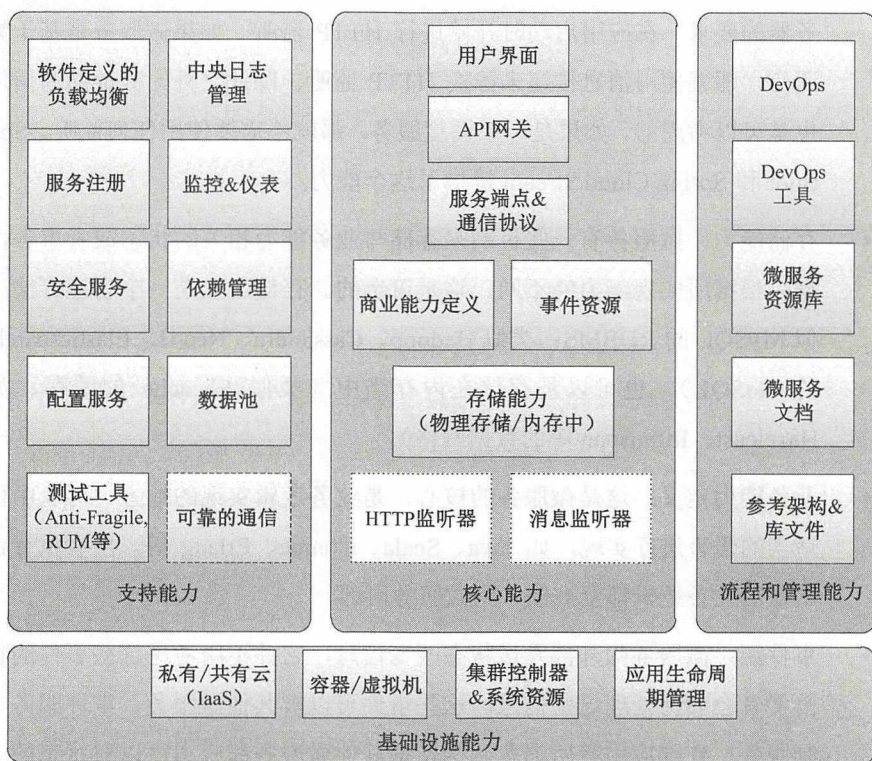
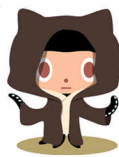


图 3-35



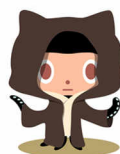
能力模型大致分为 4 个方面：

- 核心能力：这些是微服务自身的一部分。
- 基础设施能力：这是实现一个成功的微服务在基础设施级别的需求。
- 支持能力：这些是软件解决方案支持核心微服务的实现。
- 流程和管理能力：这更多的是关于流程、人员以及参考信息。

核心能力

核心能力解释如下：

- 服务监听者（HTTP/消息传递）：如果微服务支持一个基于 HTTP 的服务端点，就把 HTTP 监听嵌入到微服务中，从而消除了需要有任何外部应用服务器的要求。在应用启动时开始进行 HTTP 监听。如果微服务是基于异步通信，那就使用消息传递来替换 HTTP 监听。除此之外，其他的通信协议也是可以考虑的。如果是一个调度服务，那就不需要使用任何监听。Spring Boot 和 Spring Cloud Stream 提供了这个能力。
- 存储能力：微服务有一些机制来存储与业务能力相关的状态或者事务性数据。根据所实现能力的不同，这是可选的。存储可以是一个物理存储（类似 MySQL 的 RDBMS；类似 Hadoop、Cassandra、Neo4J、Elasticsearch 等的 NoSQL），也可以是存储在内存当中（类似 Ehcache 的缓存、类似 Hazelcast、Infinispan 等的数据网格）。
- 业务能力定义：这是微服务的核心，是业务逻辑实现的地方。可以由任何适当的语言进行实现，如 Java、Scala、Conjue、Erlang 等。所有执行任务所需的业务逻辑都会被嵌入到微服务内部。
- 事件源：微服务向外部发送状态改变信息，它并不考虑这些事件的目标消费者会如何处理这些信息。这些事件可以被其他微服务、审计服务、复制服务、外部应用等所消费。这使得其他微服务和应用可以对状态的改变做出响应。
- 服务端点和通信协议：它们定义了提供给外部消费者进行消费的 API。可以是同步端点或者异步端点。同步端点可以基于 REST/JSON 或者其他类似



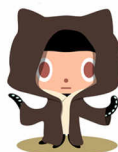
Avro、Thrift、ProtoBuffer 等协议实现。异步端点通过由 RabbitMQ 支撑的 Spring Cloud Stream、其他消息服务器或者类似如 ZeroMQ 的消息发送风格来实现。

- **API 网关:** API 网关通过代理服务端点或者组成多个服务端点提供了一个代理服务层。API 网关同样有益于策略的执行。它还提供了实时负载均衡的能力。在市场上有很多可供使用的 API 网关,如 Spring Cloud Zuul、Mashery、Apigee 及 3scale。
- **用户界面:** 通常,用户界面同样是微服务的一部分,用户利用它和微服务所实现的业务能力进行交互。用户界面可以由任何技术进行实现,它们是通道、设备无关的。

基础设施能力

毫无疑问,基础设施能力对于一个成功的部署,以及大规模的微服务管理而言是必不可少的。当部署一定规模的微服务时,不具备相应的基础设施能力会带来很大困难。

- **云:** 以传统的数据中心环境来实现微服务是相当困难的。而且为每个微服务配置大量的基础设施也是不划算的。在数据中心内部管理它们会增加拥有者的管理成本及操作成本。一个云相关的基础设施更适合用来部署微服务。
- **容器或者虚拟机:** 管理大型物理机在成本上并不划算,而且管理起来较为困难。使用物理机同样难以处理自动容错。很多组织采用虚拟机,因为它能够提供物理资源的最佳利用,还提供了资源隔离,而且还减少了管理大型物理基础设备组件的成本。容器是下一代虚拟机。VMWare、Citrix 等提供虚拟机技术。Docker、Draw 桥接、Rocket 及 LXD 是一些容器相关的技术。
- **集群控制和配置:** 一旦我们有大量容器或者虚拟机,就很难自动管理及维护它们。集群控制工具在容器的顶层提供了统一的操作环境,并且在多个服务之间共享可用的能力,如 Apache Mesos 和 Kubernetes。
- **应用生命周期管理:** 应用生命周期管理工具有助于在一个新的容器启动时唤醒应用或者在容器关闭时关闭应用。应用生命周期管理允许脚本应用的



部署和发布。它自动检测失败的场景并且对这些失败场景做出响应以便维持应用的可用性。它与集群控制软件一起工作。Marathon 一定程度上提供了这种能力。

支持能力

支持能力并不直接与微服务挂钩，但是对于大规模微服务开发是必不可少的。

- 软件定义的负载均衡器：负载均衡器应该足够智能以便能够理解部署拓扑并且做出对应的响应。它能够摒弃传统的配置静态 IP 地址、域别名，或者在负载均衡器中配置集群地址的方法。当有新的服务器加入到环境中，它应该能够自动检测到，并且在不使用任何人工手段的情况下自动将其添加到逻辑集群中。同样地，如果一个服务实例不可用，应该能够将其从负载均衡器中移除。Ribbon、Eureka 和 Zuul 配合使用能够在 Spring Cloud Netflix 中提供负载均衡能力。
- 中央日志管理：本章前面部分曾提到，需要提供把所有服务实例产生的日志附上相应 ID 集中收集的能力。这样做有助于调试、识别性能瓶颈及预测分析。将其结果反馈到生命周期管理器以采取纠错措施。
- 服务注册：服务注册提供了一个运行时环境，以便服务在运行时能够自动告知外部它们已经可供使用。从任何角度看，注册都是一个用来理解服务拓扑的不错的信息源。Spring Cloud 的 Eureka、Zookeeper 及 Etcd 是可以使用的几个服务注册工具。
- 安全服务：一个分布式微服务生态系统需要一个中央服务器来管理服务的安全。这个需要服务鉴权和 token 服务。基于 OAuth2 的服务被广泛应用于服务安全领域。Spring Security 和 Spring Security OAuth 都可以保证安全。
- 服务配置：所有服务配置都应该体现在十二因素应用原则之中。为所有配置提供一个中央服务是个不错的选择。Spring Cloud Config 服务器及 Archaius 都是开箱可用的配置服务器。
- 测试工具 (anti-fragile、RUM 等)：Netflix 使用 Simian Army 作为 anti-fragile 测试。成熟的服务需要一致的挑战来检验服务的可靠性及反馈机制的效果



如何。Simian Army 组件模拟很多错误的场景用来检验系统在失败场景下的表现。

- 监控和仪表盘：微服务同样需要一个很强的监控机制，不仅在基础设施的级别，同样需要在服务级别。Spring Cloud Netflix Turbine、Hystrix Dashboard 等类似工具提供服务级别的信息。端到端监控工具，如 AppDynamic、New Relic、Dynatrace 及其他类似 statd、Sensu、Spigo 都有助于微服务的监控。
- 依赖和 CI 管理：我们还需要工具来发现运行时拓扑、服务依赖及管理配置项，如基于图的 CMDB 等。
- 数据池：本章前面提到过，我们需要有一个机制来把不同服务之间存储的数据联系起来，并且进行实时分析。一个数据池是达到这一目标不错的选择。数据采集工具如 Spring Cloud Data Flow、Flume 及 Kafka 都可以用来消费数据。HDFS、Cassandra 等类似工具可以用来存储数据。
- 可靠消息传递：如果是异步通信方式，我们可能需要一个可靠的消息传递基础设施服务，如 RabbitMQ。云消息传递或者把消息传递作为一个服务，在互联网级的基于消息的服务端点是很流行的一种方式。

流程和管理能力

最后是微服务所需的流程和管理能力，如下：

- DevOps：开发成功的微服务的关键点是采取 DevOps。DevOps 通过支持敏捷开发、高速率传输、自动化及更好地改变管理来助力微服务的开发。
- DevOps 工具：用于敏捷开发、持续集成、持续交付及持续部署的 DevOps 工具对于微服务的成功交付是必不可少的。自动化操作、真实的用户测试、综合测试、集成、发布和性能测试等都需要着重注意。
- 微服务仓库：一个微服务仓库是微服务有版本的二进制文件存放的地方。可以是一个简单的 Nexus 仓库或者一个容器仓库，如一个 Docker Registry。
- 微服务文档：给所有微服务都编写好合适的文档是很重要的。Swagger 或者 API Blueprint 都可以用来编写微服务文档。



- 参考体系结构和库：参考体系结构在组织层面提供了一个蓝图，确保服务是根据一个确定的标准和指南、以一致的方式进行开发。很多体系结构可以被构建为一些可重用的库来强制执行服务开发理念。

总结

在这个章节，您学习了如何处理在微服务开发中会出现的实际场景。

您学习了很多解决选项和方式，能够用来解决常见的微服务问题。我们回顾了开发大规模微服务过程中可能出现的一系列挑战，以及如何有效地迎接这些挑战。

我们同样为基于微服务的生态系统建立了一个能力参考模型。这个能力模型有助于解决互联网级的微服务开发问题中出现的阻碍。本章中学习到的能力模型是本书的主干，本书的余下部分会深入学习这个能力模型。

在第 4 章中，我们会引入一个现实的问题，然后使用微服务架构将其建模，来了解如何将我们学到的知识应用到实际场景中。

第 4 章

微服务的演变 ——一个案例的学习



类似于 SOA，一个微服务可以被不同的组织根据所遇到问题的不同来进行不同的实现。除非对一个大小确定的现实问题进行详细研究，否则微服务的概念是很难深刻理解的。本章将会介绍一个虚构的航空公司——BrownField 航空（BF），以及他们从一个单体客运销售和服务（PSS）应用到下一代微服务架构的历程。本章我们将秉承上一章解释的原则和实践，详细检验这个 PSS 应用，并说明它遇到的挑战、实现方式，以及从单体系统到基于微服务架构的转换步骤。本案例研究的目的是让我们尽可能接近现实场景，以便加固我们的架构概念。

本章的最后，您会学习到以下内容：

- 一个整合单体应用到基于微服务的真实案例，通过 BrownField 航空的 PSS 应用作为例子。
- 把单体应用迁移到微服务的各种方法和过渡策略。
- 使用 Spring 框架组件来设计一个未来的微服务系统来替换 PSS 应用。

回顾微服务能力模型

本章的案例探讨了以下这些微服务能力模型，它们来自我们在第 3 章应用微服务概念所讨论的那些微服务能力模型，如图 4-1 所示。

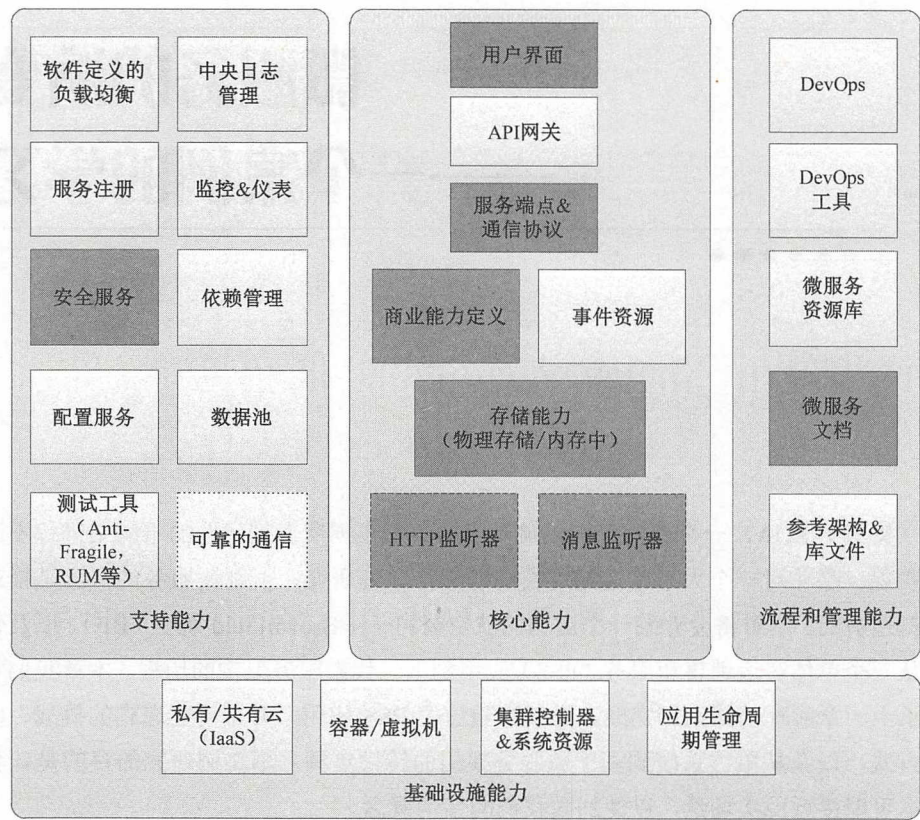


图 4-1

- HTTP 监听
- 消息监听
- 存储能力（物理/内存）
- 业务能力定义

- 服务端点&通信协议
- 用户接口
- 安全服务
- 微服务文档

在第2章“用 Spring Boot 构建微服务”里面，我们分别探讨了所有的隔离能力，包括如何保护 Spring Boot 微服务。我们会在这一章基于一个现实当中的案例学习如何创建一个综合的微服务。

提示

本章所有的源代码可在第4章项目的源码文件下获取。

理解 PSS 应用

BrownField 航空是成长最快的低成本、区域性航空公司之一，从它的中心直飞的目的地超过 100 个。作为一个刚起步的航空公司，BrownField 开始运营时只有少量的目的地和飞机。BrownField 开发其自主研发的 PSS 应用程序来处理他们的乘客销售和服务。

业务流程图

为了方便讨论，这个用例已经进行了极大的简化。图 4-2 所示的流程图展示了 BrownField 航空通过现在的 PSS 方式进行的端到端乘客服务操作。



图 4-2

目前的解决方案是自动化处理所面临的某些面向客户的功能及某些面向内部的功能。有两个面向内部的功能，飞行前和飞行后。飞行前功能包括计划阶段，用来

准备飞行调度、计划、航班等。飞行后功能提供给后方办公室用来进行收入管理、核算等。搜索和订票功能是在线座位预订流程的一部分，登记功能是在机场接收乘客的过程。登记功能同样适用于终端用户进行在线登记。

上图展示的箭头前方的叉号表示它们是断开连接的，而且发生在不同时间。例如，乘客可以提前 360 天进行预订，然而登记通常发生在飞机出发前的 24 小时内。

功能视图

图 4-3 展示了 BrownField 航空的功能性模块。每个业务流程和与之相关的子功能在同一行展示。

搜索功能	<div>搜索 指定日期和城市间的可用航班列表</div>	<div>航班 飞行航线，机型及时间计划</div>	<div>票价 城市间各航班&日期的票价</div>		
订票功能	<div>订票 选定航班&日期并订票</div>	<div>席位 选定航班&日期可用席位</div>	<div>支付 在线支付的支付网关</div>		
登记功能	<div>登记 接受旅游当日某一航班的乘客</div>	<div>登记 将乘客标识已登机</div>	<div>入座 按规定给乘客分配座位</div>	<div>行李 接受乘客行李并打印包裹标签</div>	<div>积分 更新乘客积分</div>
后台办公室功能	<div>CRM 客户关系管理</div>	<div>数据分析 商业智能分析与报表</div>	<div>税收管理 按预计的费用计算</div>	<div>财会 发票与账单</div>	
数据管理功能	<div>参考数据 国家、城市、航班、货币等</div>	<div>客户 管理客户</div>			
横切功能	<div>用户管理 管理用户、角色、特权</div>	<div>通知 发送短信和邮件给客户</div>			

图 4-3

图 4-3 中展示的子功能展现了它在整个业务流程中所扮演的角色。有些子功能参与到了多个业务流程。例如，库存同时用于搜索和订票。为了避免复杂，并未在图中展示这一点。数据管理和跨领域子功能在多个业务流程中使用。

架构视图

为了更有效率地管理端到端乘客操作，大约十年前，BrownField 开发了一个内部 PSS 应用。这个架构优良的应用是使用 Java 和 JavaEE 技术及在当时所能提供的

最好的开源技术进行开发的。总体架构和技术展示在图 4-4 中。

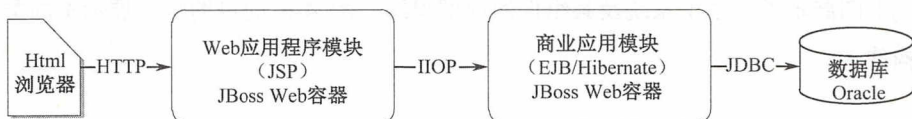


图 4-4

这个架构有清晰的边界。同时，不同的关注点分离到不同的层。这个应用被开发成为一个 N 层、基于组件的模块化系统。相互之间的功能交互是通过以 EJB 端点形式进行定义的服务契约。

设计视图

这个应用有很多逻辑功能组或者子系统。而且，正如图 4-5 所描绘的，每个子系统有很多有组织的组件。

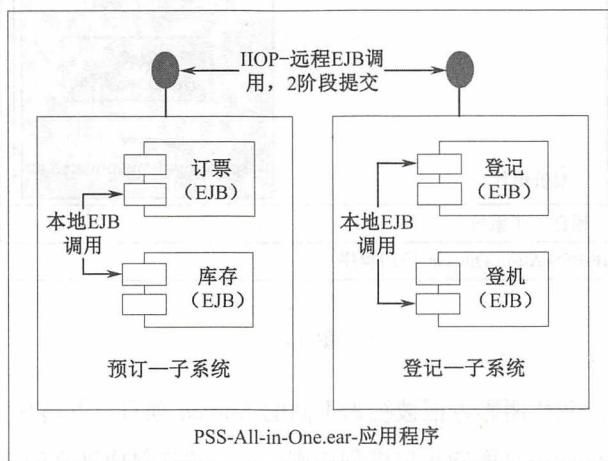


图 4-5

子系统相互之间是使用 IIOP 协议通过远程 EJB 调用进行交互的。事务边界跨子系统。子系统内部组件相互之间通过一个本地 EJB 组件接口进行通信。理论上，由于子系统使用远程 EJB 端点，它们可以运行在物理上分开的应用服务器上。这也是其中一个设计目标。

实现视图

下图展示了一个子系统及其组件的内部组织。图 4-6 的目的也是展示不同类型的构件。

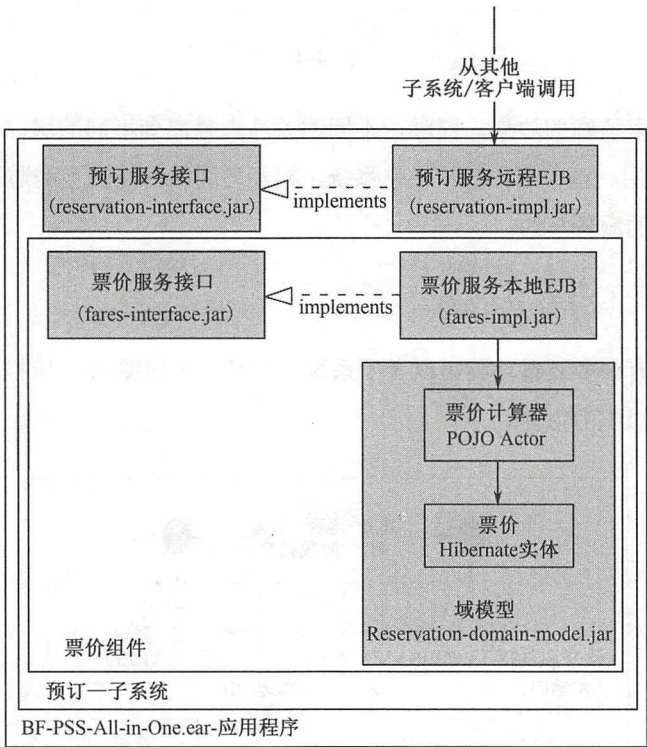


图 4-6

在图 4-6 中，灰色阴影方框被视为不同的 Maven 项目，并转化为物理构件。子系统和组件被设计成符合面向接口编程原则。接口被打包成独立的 jar 文件，以便客户端从实现中抽象出来。业务逻辑的复杂性被掩埋在域模型中。本地 EJB 作为组件接口。最后，所有子系统被打包成一个一体化 EAR，并被部署在应用服务器中。

部署视图

应用的原始部署是简单并且直接的，图 4-7 展示了这一点。

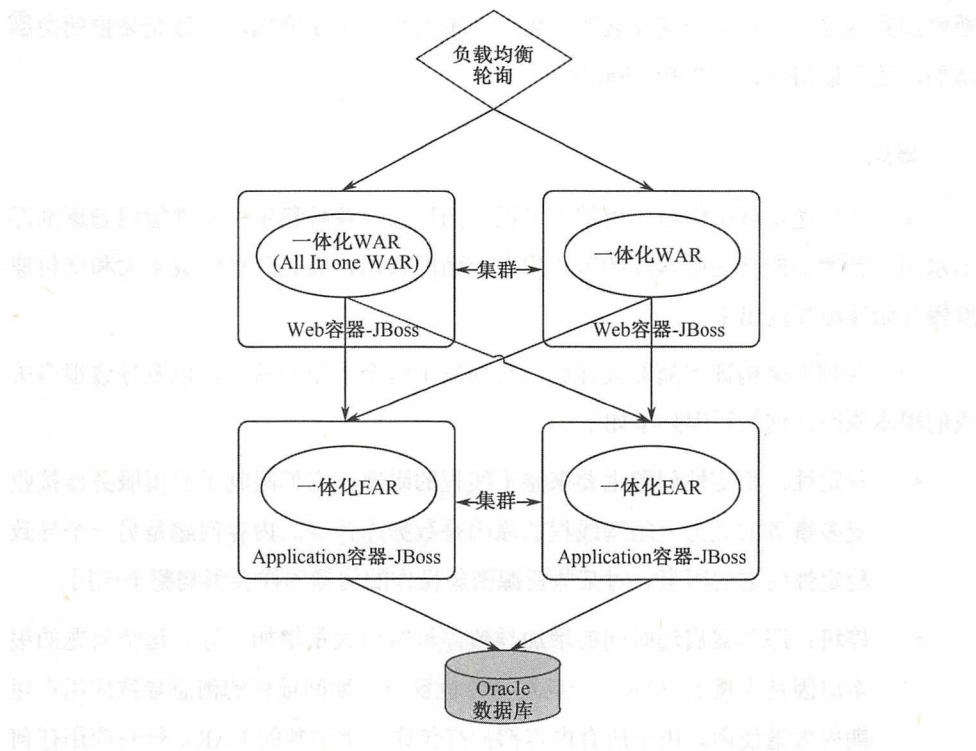


图 4-7

Web 模块和业务模块被分开部署在应用服务器集群中。通过向集群中不断增加应用服务器，应用可以进行横向扩展。

通过创建一个备用集群，然后把流量平滑地转向那个集群，即可实现零停机部署。一旦主集群使用新版本进行补丁并且影响了服务，备用集群即被销毁。大多数数据库被设计为具有向后兼容性，但是重大改变促使应用中断。

庞然大物的终结

PSS 应用表现得很不错，成功支撑了所有业务需求及所预期的服务级别。在业务自然增长的最初几年里，这个系统表现的没有任何问题。

一段时间过后，业务出现了爆发式增长。发展的脚步大步迈开，新的目的地不

断增加到网络中。作为快速发展的结果，订单的数量不断增加，导致交易量的急剧增加，达到最初估计的 200~500 倍。

痛点

业务的快速发展最终给应用带来了很大的压力，各种稳定性和性能问题逐渐浮出水面。新增加的功能版本开始破坏生产上面的代码，而且变更的成本大和交付速度慢开始逐渐显露出来。

一个端到端架构评审随即被开展，它暴露了这个系统的缺陷，以及导致很多失败的根本原因，这些原因列举如下：

- 稳定性：稳定性问题主要来源于线程的阻塞，它们限制了应用服务器接收更多事务的能力。阻塞线程的原因是数据库锁表。内存问题是另一个导致稳定性问题的因素。对某些资源密集操作的问题同样会影响整个应用。
- 停机：服务器启动时间的增加导致停机窗口大量增加。导致这个问题的根本原因是大量的 EAR。一旦发生停机窗口，期间堆积的消息导致应用在短期内大量使用。由于所有内容都被打包成一个单独的 EAR，针对应用任何小的代码的改动都需要对整个应用重新进行部署。前面提到的零宕机部署模型的复杂性及服务器启动时间的增加，共同增加了停机数量及持续的时间。
- 敏捷性：随着时间的推移，代码也变得越来越复杂，这一点部分来源于在对代码进行变更的时候缺乏统一的规范。这最终导致了更难以去维护这些代码。同时也很难去调查问题或者不能确定问题的最佳解决方法。最终不确定的解决方法就不能有效地去修复生产代码。应用构建时间大量增加，从几分钟到几个小时，导致开发效率急剧下降。构建时间的增加同时导致难以进行自动化构建，最终阻碍了持续集成（CI）和单元测试。

权宜之计

我们在第 1 章“解密微服务”中曾提到过，部分性能问题可以通过在规模数据集中使用 y 轴扩展方法得以解决。包罗万象的 EAR 都被部署到多个不相交集群。使用安装好的软件代理将流量有选择地路由到指定集群，如图 4-8 所示。

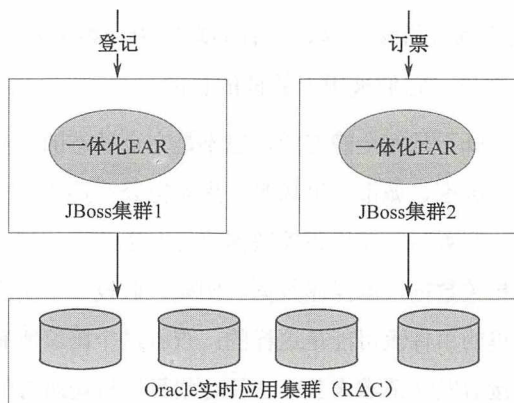


图 4-8

这有助于 BrownField 的 IT 人员扩展应用服务器。由此，稳定性问题得以控制。但是，这个随后在数据库级别引发了一个瓶颈。Oracle 开发了实时应用集群（Real Application Cluster，RAC）用于在数据库层解决这一问题。

这一新的扩容模型降低了性能问题，但是相应地增加了系统的复杂性和降低了微服务自身价值。在一定时间内也增加了技术债务，以致完全重写是减少这种技术债务唯一的选择。

反思

尽管应用有着优秀的架构，并且在功能组件之间有着明显的隔离。它们是松耦合的、面向接口编程的，并且通过基于标准的接口访问，还有着丰富的域模型。

一个很明显的问题是，这么一个有着优秀架构的应用，为何达不到预期？架构师还能怎么做？

理解在一定时间内是哪里出了错误是很重要的。就本书而言，最为重要的一点在于理解微服务是如何避免这些场景再次发生的。我们将在随后的章节中研究其中的一些场景。

共享数据

几乎所有功能性模块都需要参考类似航线的详情、飞机详情、机场和城市列表、国家等数据信息。例如，机票是基于起点（城市）、航班是在起点和终点（机场）之

间、登记发生在起点机场（机场）等。在有的功能中，参考数据是信息模型的一部分，然而在另一些功能里，它们被用于验证的目的。

这些参考数据大多既不是完全静态的，也不是完全动态的。当航空公司新开一条航线时才可能新增一个国家、城市、机场等。当航空公司购买了一个新的飞机，或者修改了已有飞机的座位配置时，飞机相关数据就会发生改变。相关数据的一个常见使用场景是，基于某些相关数据过滤操作数据。例如，假设一个用户想要查看去往一个国家的所有航班。这里的事件流可能是这样的：查看这个国家所有的城市，然后是这个城市所有的机场，接着发送请求来获取去往这些目标机场所有的航班列表。

架构师在设计这个系统时考虑了多种方案。其中一个方案是，就像其他子系统那样，把这些相关数据分开成为一个个独立的子系统，但是这会带来性能问题。不同于其他事务，团队采取了遵循一个异常的方法来处理这些相关数据。考虑到前面所讨论的查询模式的性质，这个方案是把相关数据作为一个共享库。

在这种情况下，子系统可以使用引用传递语义数据直接访问参考数据，而不是通过 EJB 接口。这也意味着，除了子系统，Hibernate 实体也可以使用相关数据作为它们实体关系的一部分。

如图 4-9 所描述，在预约子系统订票子系统可以使用本例中机场实体的相关数据来作为它们关系的一部分。

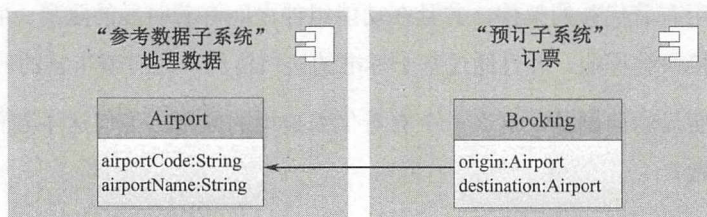


图 4-9

单一数据库

尽管在中间层执行了足够的隔离，所有的功能都指向了同一个数据库，甚至同一个数据库表。这种单一模式的方式带来了一大堆问题。

原生查询

Hibernate 框架对底层数据库做了良好的抽象。它产生了高效的 SQL 语句，在大多数情况下针对目标数据库使用对应的语句。但是，有时编写原生 JDBC SQL 可以带来更好的性能和资源效率。在有些情况下，使用原生数据库函数提供更好的性能。

单一数据库方案在初期可以正常使用。但是一段时间之后，它带来了一个漏洞，开发人员可以通过不同的子系统连接数据库表。原生 JDBC SQL 就是做这件事的一个很好的工具。

图 4-10 展示了使用一个原生 JDBC SQL 连接两个分属于不同子系统的数据库表的例子。

如图 4-10 所示，账单组件需要从订票组件获取同一天内某个城市所有的订票记录来进行当日账单结算。基于子系统的设计强制账单组件发送一个服务请求到订单组件来获取给定城市所有的订单记录。假设这个记录是 N 。现在，对于每一条订票记录，账单服务执行一个数据库请求，基于每条订票记录上的费用代码来发现合适的规则。这需要 $N+1$ 次 JDBC 请求，显然这是低效的。稍微变通一下，如使用批量或者并行查询和批量执行，但这会带来编码量的增加和复杂度的提升。开发人员通过简化原生 JDBC 查询来解决这一问题。本质上，这种方式可以使用最少的代码量把 $N+1$ 次查询请求减少到单个数据库请求。

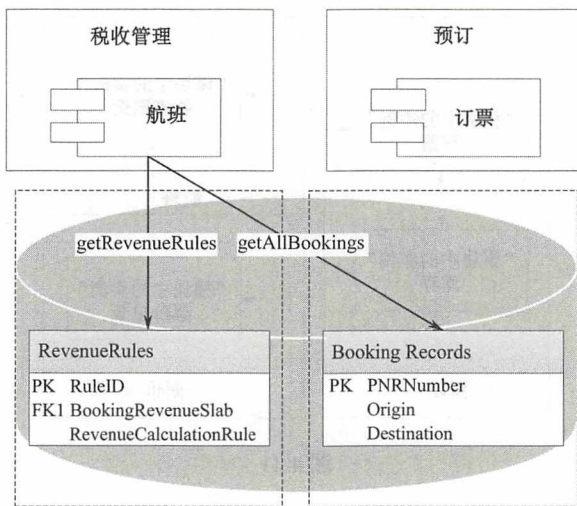


图 4-10

在许多 JDBC 本地查询连接跨多个组件和子系统表的情况下都会有这种操作习惯。但这不仅会导致组件紧密的耦合，还会带来没有文档、难以测试的代码。

存储过程

使用单一数据库还会带来的一个问题是使用复杂的存储过程。编写在中间层的很多复杂的核心逻辑都表现得不好，导致响应缓慢、内存问题以及线程阻塞问题。

为了解决这个问题，开发人员决定把部分复杂的业务逻辑从中间件层移到数据层，通过直接在数据库的存储过程内部实现这些逻辑。这个决定给有些业务带来了更好的性能并且避免了一些稳定性的问题。一定时间内，大量的存储过程被添加进去。但是，这会破坏应用的模块性。

域边界

尽管域边界被很好地创建，所有的组件都被打包成单个 EAR 文件。由于所有组件都被设置为运行在单个容器内，并不能阻止开发人员引用不同边界里的对象。经过一段时间，项目组发生了改变，交付压力随即上升，复杂度惊人地增加。开发人员开始寻找快速的而不是正确的解决方案。慢慢地，应用的模块化性质就不见了。

如图 4-11 所示，在多个子系统边界中产生了 Hibernate 关联。

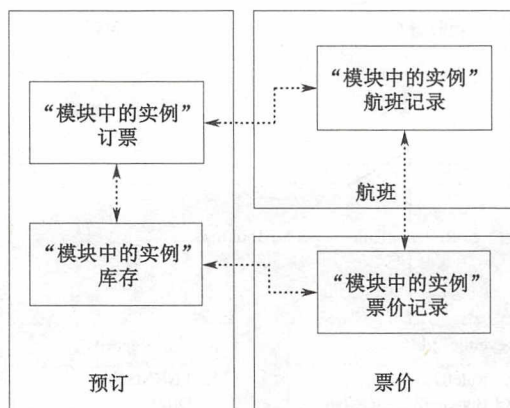


图 4-11

使用微服务来拯救

没有太多改进机会留给 BrownField 来支撑其不断增长的业务需求。BrownField 航空寻求使用一个演化方式而不是一个革命性模型来重新制定系统的平台。

微服务是这些情况下的理想选择——使用对业务影响最小的方式对遗留的单体应用程序进行转变。

如图 4-12 所示，目标是在保持同样业务能力的前提下改造成基于微服务的架构。每个微服务都包含数据存储、业务逻辑及表现层。

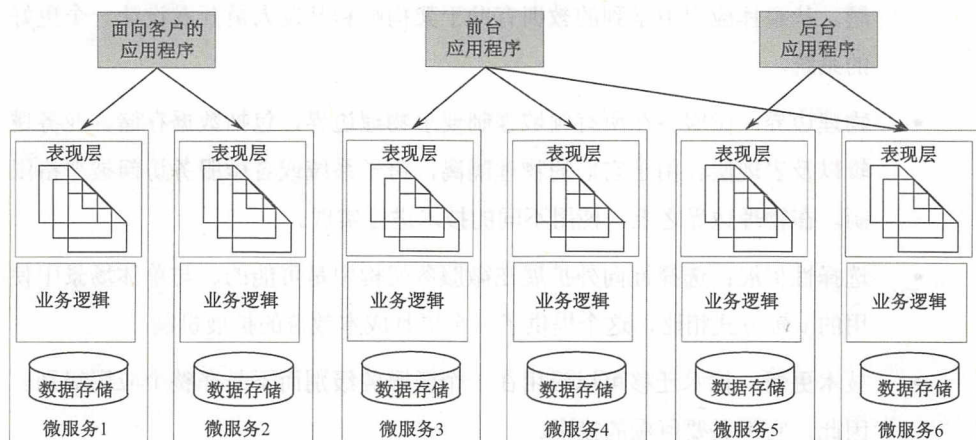


图 4-12

BrownField 航空采取的方式是，针对特定的用户群体，如面向客户、前台、后勤等，创建一系列 Web 门户应用。这种方式的优势在于建模的灵活性和能够区别对待不同的用户群体。例如，面向互联网的策略、架构和测试方法与面向内部网络的 Web 应用是有区别的。面向互联网的应用可能利用 CDN（Content Delivery Networks：内容分发网络）来使页面尽可能地接近用户，然而内部网络应用可以直接从数据中心提供页面。

业务用例

在为了迁移创建业务用例时，一个最常问的问题是“微服务架构在接下来的 5 年时间里如何避免出现同样的问题？”

微服务提供了诸多好处，这在第 1 章“解密微服务”中已经学习过了，但还是很有必要在这里列举一部分。

- 服务依赖：在把单体应用向微服务迁移过程中，最好提前弄清楚依赖情况，以便架构师和开发人员能够更好地避免打破依赖关系和可以预见的依赖问题。从整体应用中学到的教训有助于架构师和开发人员开发设计一个更好的系统。
- 物理边界：微服务在所有领域强制要求物理边界，包括数据存储、业务逻辑以及表现层。由于它们的物理隔离，跨子系统或者微服务访问被严格限制。在物理边界之上，使用不同的技术来实现。
- 选择性扩展：选择性向外扩展在微服务架构中是可能的。与单体场景中使用的 y 轴方式相比，这个提供了一个更具成本效益的扩展机制。
- 技术更新：技术迁移可以应用在一个微服务级别而不是在整个应用级别。因此，它不需要巨额的投资。

为演化制订计划

把一个拥有百万行代码的应用进行切分并不简单，尤其是当代码有复杂的依赖时。那我们如何进行切分？更为重要的是，我们从哪里开始，以及我们如何解决这个问题？

演化方法

解决这个问题最好的办法是建立一个过渡计划，然后逐渐把功能迁移为微服务。

在每一步中，一个微服务会在这个单体应用外部创建，流量会被转移到新建的服务，图4-13展示了这一点。

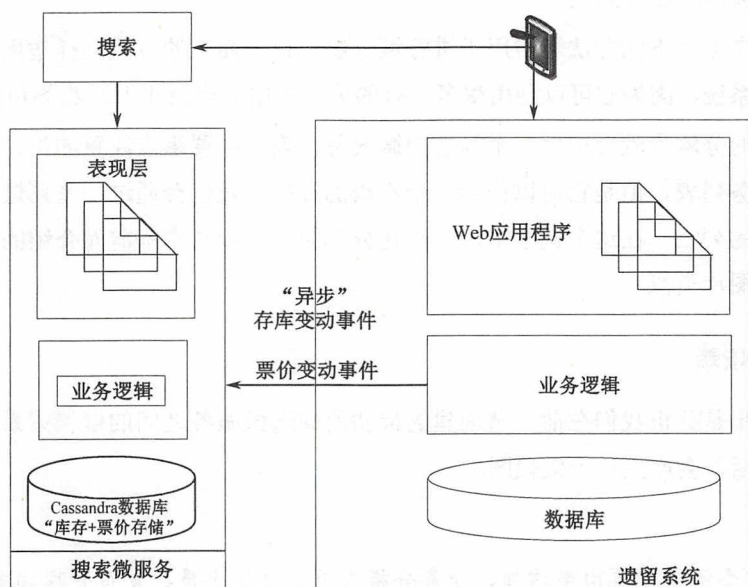


图4-13

为了进行成功的迁移，一些关键问题需要从转型角度进行回答。

- 识别微服务边界。
- 微服务迁移的优先级。
- 在转变过程中处理数据同步。
- 处理用户接口迁移，使用老的和新的用户接口。
- 在新系统中处理参考数据。
- 测试策略，确保业务功能是完整的并被正确的复制了。
- 识别微服务开发中的先决条件，如微服务能力、框架、流程等。

识别微服务边界

首先，也是最重要的任务是确定微服务的边界。这是向微服务迁移中最有趣的部分，同时也是最困难的部分。如果对边界的识别没有恰当的进行，迁移过程会导致更加复杂的管理问题。

就像在 SOA 中那样，识别服务最好的方法是对服务进行分解。但是，重要的是注意把分解停留在业务能力或者有界上下文中。在 SOA 中，服务分解深入到一个原子的、细粒度的服务级别。

一个自上而下的方法通常用于进行域分解。自下而上的方式同样适用于分解一个已有的系统，因为它可以利用很多已有的单体应用的实践知识、业务和行为。

前面的分解步骤会产生一个候选的微服务列表。需要重点注意的是，这并非最终的微服务列表，但是它可以作为一个不错的开始。我们会通过一系列过滤机制来得到最终的列表。在这个例子中，功能化分解的第一步与本章前面介绍的功能视角下展示的图片类似。

分析依赖

第二步是分析我们在前一节创建的最初的候选微服务之间的依赖关系。在这项活动的最后，会产生一个依赖图。

提示

这个活动需要由架构师、业务分析人员、开发人员、发布管理和支持人员组成的团队来完成。

产生一个依赖图的一种方式列举遗留系统所有的组件和所覆盖的依赖。这个可以通过配合以下列举的方法中的一种或多种进行实现。

- 分析手工代码然后生成依赖关系。
- 根据开发团队的经验生成依赖关系。
- 使用一个 Maven 依赖图。有很多工具可以用于产生依赖图，如 PomExplorer、PomParser 等。
- 使用性能工程工具比如 APPDynamic 来识别调用栈，然后分析依赖关系。

假设我们复现了如图 4-14 所示的功能和依赖关系。

有很多依赖在不同的模块反复出现。底层展示了用于跨模块的交叉功能。在这一点上，模块更像是意大利面而不是自治单元。

下一步是分析这些依赖，然后得到一个更好的、简化的依赖图。

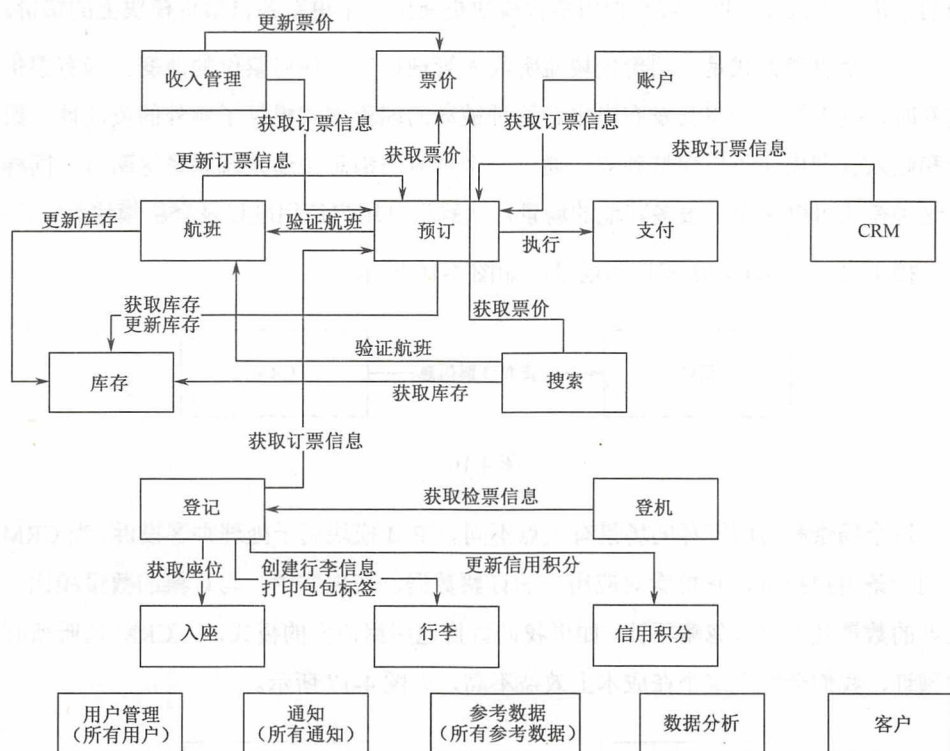


图 4-14

基于事件而非查询

依赖可以是基于查询的或者基于事件的，可伸缩系统更适合基于事件的方式。有时，可以把基于查询的通信转化为基于事件的方式。很多情况下，之所以会产生这些依赖，要么是因为业务组织是这样管理的，要么是因为旧系统依靠这种方式处理业务场景。

在图 4-14 中，我们可以提取收入管理和票价服务，如图 4-15 所示。

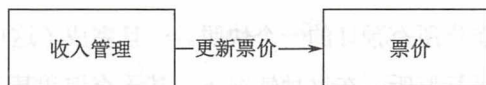


图 4-15

收入管理是用于计算最优值的模块，基于订票需求预测。如果在起点和终点之

间的票价发生改变，收入管理调用票价模块更新票价来更新各自票价模块上的票价。

另一个思考方式是，票价模块监听收入管理以应对任何票价的变化。当有票价改变时，收入管理即将其发布出来。这种被动的编程方式提供了额外的灵活性，票价和收入管理模块可以保持独立，通过一个可靠的消息传递系统联系这两者。同样的这种模式可以应用于很多其他的场景，从登记到用户信用度以及登机模块。

接下来，考虑 CRM 和订票场景，如图 4-16 所示。

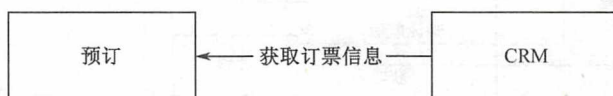


图 4-16

这个场景和前面解释的场景有一点不同。CRM 模块用于处理乘客投诉。当 CRM 收到一条用户投诉，它检索对应用户的订票数据。在现实中，与订票的数量相比，投诉的数量几乎可以忽略不计。如果我们盲目地应用前面的模式，让 CRM 监听所有的预订，我们会发现这个在成本上效益不高，如图 4-17 所示。

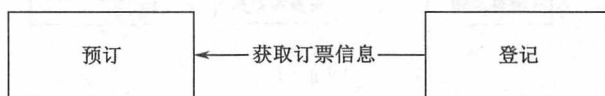


图 4-17

考虑存在于登记和预订模块之间的另一个场景。能否使用登记模块监听预订事件这种方式来替换登记模块在预订时调用 Get Bookings 服务的方式？这是有可能的，但是存在一个问题，预订可以提前 360 天发生，但是登记通常只在飞机起飞的前 24 小时开始。提前 360 天在登记模块中重复所有预订和预订更改并非一个明智的决定，因为登记模块在飞机起飞前的 24 小时之外都不需要这些数据。

另一个选择是当一个航班开始登记（起飞前的 24 小时），登记调用预订模块的一个服务来获取给定航班所有预订的一个快照。一旦完成了这项工作，登记模块就可以只针对那个航班进行监听。在这种情况下，基于查询和基于事件的方式被配合使用。通过这么做，除了减少了这两个服务间的查询数量，我们还减少了非必要的事件和存储。

简而言之，没有一个策略适合所有的场景。每个场景都需要进行逻辑思考，然后应用最合适的模式。

使用事件而非同步更新

除了查询模型，一个依赖也可以是一个更新事务。考虑收入管理和预订之间的场景，如图 4-18 所示。

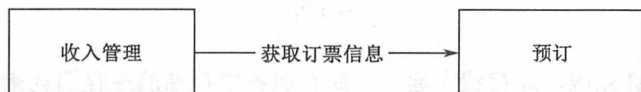


图 4-18

为了对当前的需求做出预测和分析，收入管理需要所有航班的所有预订。在依赖图中展示的当前方式是，收入管理有一个调度作业，在预订服务中调用 GetBookings 来获取最后一次同步后所有增加的预订（新增的和修改的）。

另一种方式是在预订中发送新的预订和改变，一旦它们在预订模块中作为异步推送所产生。同样的模式可以应用于很多其他场景中，如从预订到记账、从航班到库存，以及从航班到预订。在这种方式中，源服务发送所有状态改变的事件到一个主题中。所有对此感兴趣的服务可以监听这个事件流然后存储到本地。这种方式移除了很多硬编码，保持了系统的松耦合。

图 4-19 展示了依赖关系：

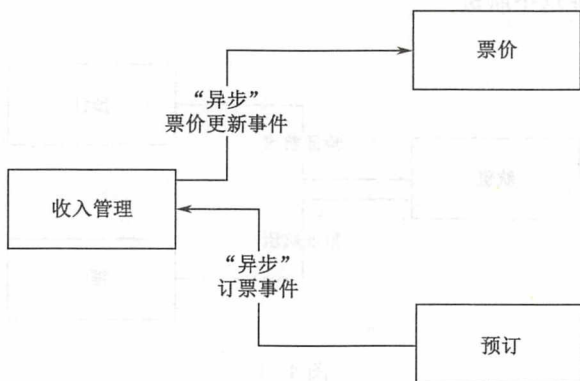


图 4-19

在图 4-19 所示的例子中，我们同时改变了两侧的依赖并且把它们转化成异步事件。

最后一个要分析的例子是预订模块对库存模块发起的更新库存调用，如图 4-20 所示。

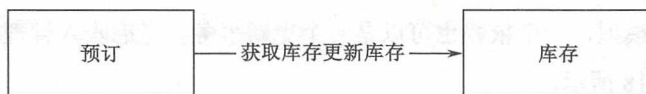


图 4-20

当一个预订完成，库存状态通过在库存服务中存储的库存消耗来进行更新。例如，当有 10 个商务座可供使用，预订发生后，座位减少为 9 个。在当前的系统中，预订和更新执行在同一个事务边界。这是为了解决一个场景，当前只剩一个座位，多个用户都想要进行预订。在新的设计中，如果我们应用同样的事件驱动模式，把库存更新作为一个事件发送给库存服务，可能让系统处于一个不一致的状态。这个需要进一步分析，我们会在本章的后面进行阐述。

挑战需求

在很多例子中，可以通过采取重新审视需求来达到目标状态。

如图 4-21 所示，这里有两个验证航班请求，一个来自预订模块，另一个来自搜索模块。验证航班请求是用来验证从不同渠道输入的航班数据。最终的目标是为了避免错误的存储或服务数据。当一个用户进行航班搜索，假设是“BF100”，系统从下面几个方面验证这个航班：

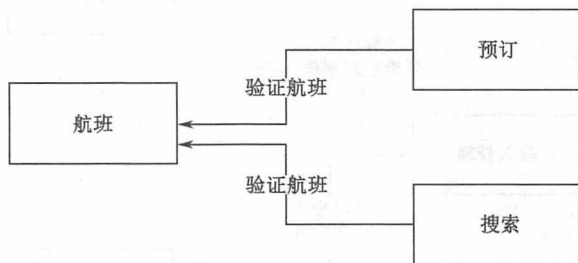


图 4-21

- 这是否是一个合法的航班？
- 这个航班在那天是否存在？
- 这个航班是否设置有任何预订限制？

另外一个解决这个问题的方式是根据这些给定的条件调整飞机的库存。例如，如果对于航班有限制，把库存更新为 0。在这个例子中，情报机关会保留这个航班，并保持更新数据库。对于搜索和预订而言，它们只需要查看库存而不用对于每个请求都验证航班。这种方式与最初的方式相比更加有效率。

接下来我们将评审支付用例。由于类似 PCIDSS 标准的安全约束的本质，支付通常是一个离线功能。捕获一个支付最明显的方式是把浏览器重定向到一个部署在支付服务上的支付页面。由于银行卡处理应用需要符合 PCIDSS 条款，把支付服务上的任何直接依赖移除是明智的。因此，我们可以移除预订到支付的直接依赖，然后选择在 UI 层集成。

挑战服务边界

在这一节，我们会根据需求和依赖图评审一些服务边界，考虑登记功能和它对座位服务和行李服务的依赖。

座位功能根据当前飞机上座位的分配来运行一些算法，然后选择最好的方式来给下一位乘客分配座位，以便可以满足重量和平衡的需求。这个是基于事先定义的业务规则。但是，除了登记，没有其他模块对座位功能感兴趣。从一个业务能力角度，座位只是登记的一个功能，它本身并非一个业务能力。因此，更好的做法是把这个逻辑嵌入到登记自身。

这点对于行李同样适用。BrownField 有一个独立的行李处理系统。行李功能在 PSS 中是打印行李标签和针对登记记录存储行李数据。没有业务能力与这个特定的功能相对应，因此，把这个功能移到登记本身更为理想。

图 4-22 展示了重新设计之后的预订、搜索和库存功能。

同样地，库存和搜索功能更像是预订模块的支持功能。它们不与任何类似的业务能力相关。类似于之前的判断，把搜索和库存功能移到预订中更加合适。假设暂时将搜索、库存和预订功能都移到一个名为预订的微服务中。

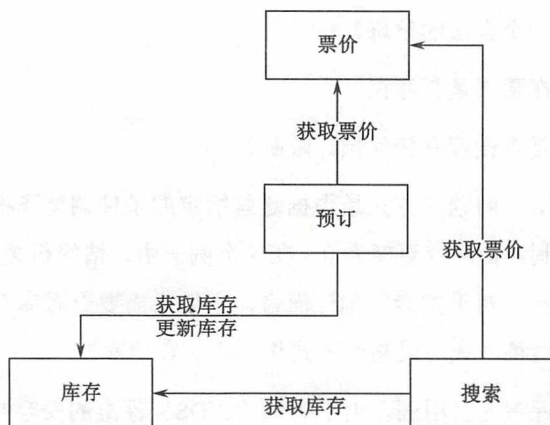


图 4-22

根据 BrownField 的统计数据,搜索事务比订票事务发生的频率频繁 10 倍。而且,相对于预订而言,搜索并非一个盈利性事务。出于这个原因,对于搜索和预订,我们需要不同的扩展性模型。如果有搜索事务的激增,不应该对预订服务造成影响。从业务的角度看,为了保护预订事务而牺牲搜索事务是可以接受的。

如图 4-23 所示,这是一个多语言需求的例子,它推翻了业务能力走向。在这个例子中,把搜索从预订服务中独立成一个微服务更有意义。假设我们移除了搜索,只有库存和订票服务保留在预约微服务中。那么搜索需要反过来请求预约微服务来查询库存,这可能影响预订事务。

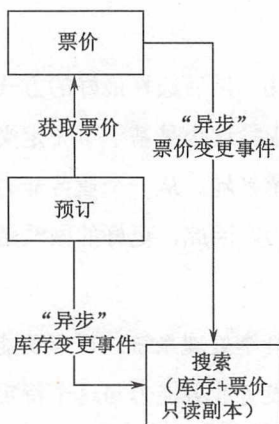


图 4-23

一个更好的方式是保留库存和订票模块,然后在搜索模块下复制一份只读的库存模块副本,通过一个可靠的消息传递系统同步库存数据。由于库存模块和订票模块是并列的,这也解决了两阶段提交的需求。由于它们都是本地的,那么可以使用本地事务正常工作。

现在让我们挑战票价模块的设计。当一个客户搜索 AB 之间某一天的航班,我们想要同时显示航班和票价,这意味着我们的只读副本的库存微服务就需要同时连

接票价服务和库存服务。搜索模块监听来自票价微服务的票价改变事件，同时票价微服务需要不断发送票价更新事件到搜索微服务。

最终依赖图

还有一些同步请求，我们暂时保留它们原来的样子。

通过应用所有这些改变，最终的依赖图会成为图 4-24 所展示的样子。

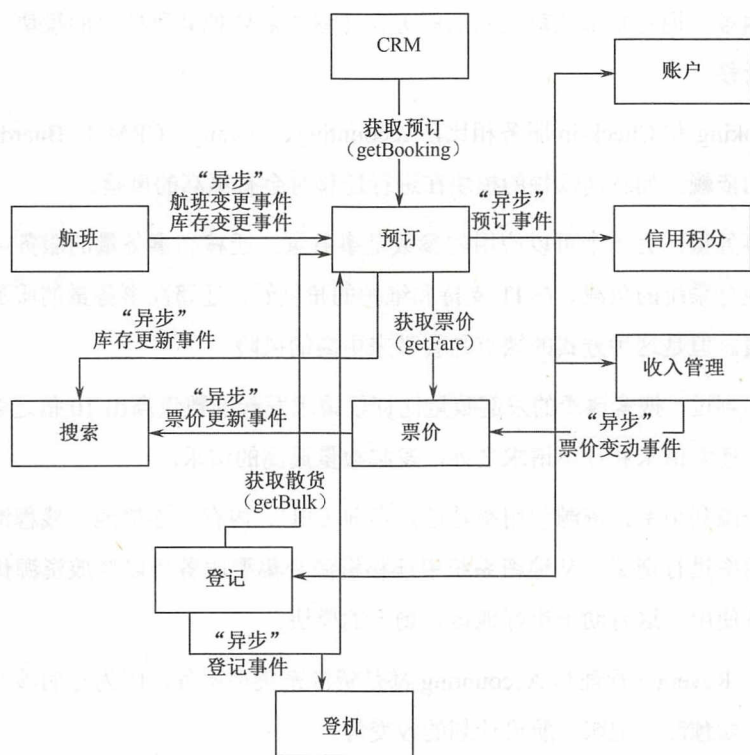


图 4-24

现在我们可以安心地考虑把图中的每个方框都作为一个微服务。我们已经确定了很多依赖，并且把它们中的一部分建模为异步。整个系统或多或少被设计成被动形式。图中用粗线标注的请求都属于同步请求，如 Check-in 服务中的 Get Bulk，CRM 中的 Get Booking，Booking 中的 Get Fare。根据权衡分析，这些请求有必要被设计成为同步的。

微服务迁移的优先级

对于我们基于微服务的架构，我们已经确定了一个粗略的版本。作为下一步，我们会分析优先级，然后确定迁移的顺序。这需要考虑以下列举的几个因素来进行决定：

- 依赖：一个决定优先级的因素是依赖图。从服务依赖图来看，拥有少量或者没有依赖的服务更容易迁移，拥有复杂依赖关系的服务迁移起来则更为困难。拥有复杂依赖关系的服务在迁移时需要把其所依赖的模块一同进行迁移。

与 Booking 和 Check-in 服务相比，Accounting、Loyalty、CRM 和 Boarding 服务有着更少的依赖。拥有高依赖的模块在进行迁移时会有更高的风险。

- 事务量：另一个可以应用的参数是事务量。迁移高事务量的服务可以减轻现有系统的负载。在 IT 支持和维护的角度看，迁移高事务量的服务更有价值。但是这种方式的缺点是会带来更高的风险。

前面提到过，搜索请求的发起数量比订票请求发起的数量高出 10 倍之多。登记请求是除了搜索请求和订票请求之外，发起数量最高的请求。

- 资源利用率：资源利用率是通过当前 CPU、内存、连接池、线程池等的利用率进行衡量。从遗留系统中迁移资源密集型服务可以释放资源供其他服务使用，这有助于更好地运行剩下的模块。

Flight、Revenue 管理和 Accounting 都是资源密集型服务，因为它们涉及数据密集型事务，如预测、记账、航班计划的改变等。

- 复杂度：复杂度可能根据功能点、代码量、数据库表的数量、服务数量等服务的业务逻辑进行衡量。与复杂模块相比，简单模块更加容易进行迁移。

与 Boarding、Searching 及 Check-in 服务相比，Booking 服务显得极为复杂。

- 业务关键性：业务关键性可以根据收入或者用户体验进行确定。越关键的服务能够带来更高的商业价值。

从商业的角度看，Booking 服务是最创收的服务，然而 Check-in 服务对于业务

也很关键，因为它会导致航班延误，这会导致收入减少和客户的不满。

- 改变的速率：改变的速率表示在短期内针对某个功能进行改变的次数。这会转化为交付的速度和敏捷度。需要经常发生改变的服务与稳定的模块相比，它们更适合进行迁移。

经过统计，Search、Booking 和 Fare 服务需要频繁改变，而 Check-in 服务是最为稳定的功能。

- 创新：属于颠覆性创新流程一部分的服务需要优先于基于更成熟的业务流程的后勤功能服务。与在微服务中应用创新相比，在遗留系统中进行创新更难实现。

大多数的创新是关于 Search、Booking、Fares、Revenue Management 和 Check-in 服务，在 Accounting 这样的后勤服务进行创新的可能性较小。

根据 BrownField 的分析，Search 服务拥有最高的优先级，因为它需要创新、拥有更高的改变率、更少的关键业务，更多地缓解了业务和 IT 的压力。Search 服务有最少的依赖，而且没有从遗留系统中同步数据的需求。

迁移过程中的数据同步

在过渡阶段，遗留系统和微服务会并行。因此，在两个系统之间保持数据同步是很重要的。

同步数据最简单的方式是通过使用数据同步工具在数据库级两个系统之间同步数据。当新、老两个系统是建立在同一种数据存储技术上时，这种方式能够很好地完成任务。如果它们所使用的数据存储技术不同，这种方式的复杂度会变高。另外，这相当于我们留下了一个后门入口，从而把微服务的内部数据存储暴露在了外面。这都是有悖于微服务原则的。

在我们能得出一个通用解决方案之前，先看看不同的方法有哪些优劣。图 4-25 展示了一旦搜索请求发出时，数据迁移和同步相关的情况。

假设我们使用一个 NoSQL 数据库来保存在 Search 服务下的库存和票价。在这种特殊情况下，我们需要遗留系统通过异步事件的形式向新的服务提供数据。我们

修改已有的系统，将票价改变或者任何库存改变作为事件进行发送。Search 服务接着接收这些事件，然后把它们存储在本地 NoSQL 数据库。

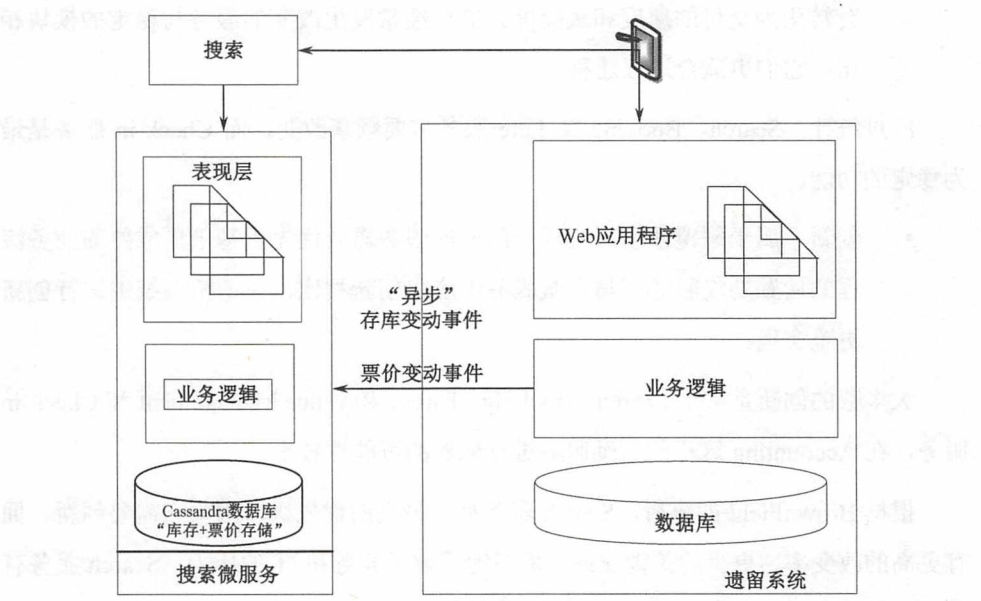


图 4-25

对于复杂的 Booking 服务来说，这个稍显烦琐。

新的 Booking 微服务发送库存改变事件到 Search 服务。除此之外，遗留应用同样需要发送票价改变事件到 Search 服务。Booking 服务接着在它的 MySQL 数据库中存储新的 Booking 服务。

接下来是最复杂的部分，Booking 服务需要发送订票事件和库存事件返回给遗留系统。这是为了保证遗留系统的功能能够像从前一样运转。最简单的方式是编写一个更新组件，它接收事件然后更新老的预订记录表，这样在其他遗留模块上就不需要进行任何改变。我们一直这样做，直到没有任何遗留组件引用订票和库存数据。这有助于我们对遗留系统的改变最小，因此，也减少了失败的风险，如图 4-26 所示。

简而言之，单一的方法是不够的，必须根据不同模式采取多管齐下的方法。

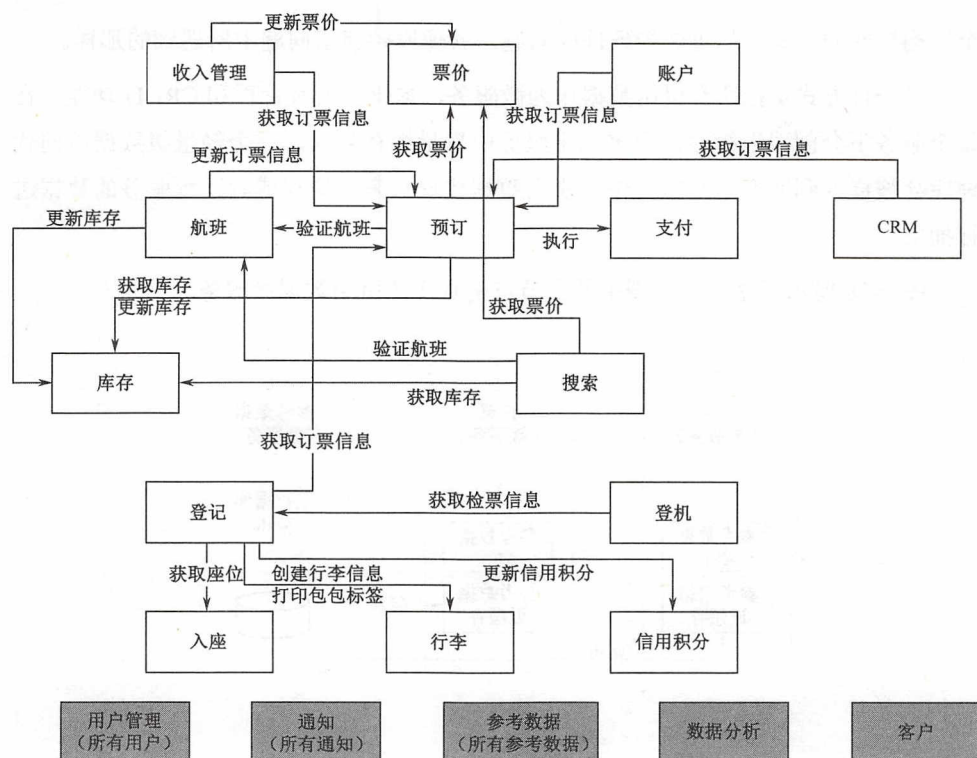


图 4-26

管理引用数据

迁移单体应用到微服务中一个最大的挑战是管理引用数据。一个简单的方法是把引用数据本身创建为另一个微服务，如图 4-27 所示。

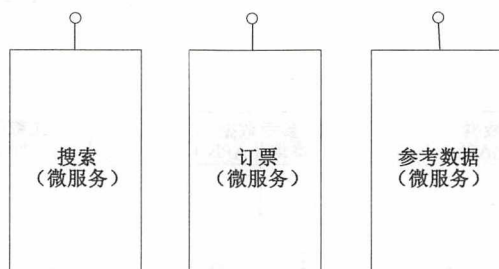


图 4-27

在这种情况下，不论谁需要引用数据，都需要通过微服务点访问它们。这是一

个结构良好的方式，但可能导致性能问题，就像原始遗留问题中所遇到的那样。

另一种方式是把所有引用数据作为微服务，提供给所有管理和 CRUD 功能。在每个服务下会创建近缓存，用来从主服务中增量缓存数据。一个轻量级数据访问代理库会被嵌入到所有这些服务中。引用数据代理对来自缓存或者远程服务的数据进行抽象。

图 4-28 展示了这一点，图中的主节点是真实的引用数据微服务。

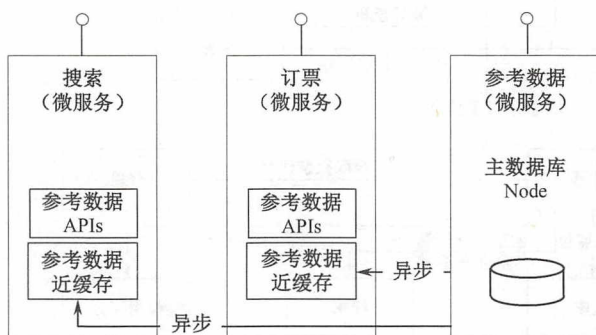


图 4-28

这里的挑战是主从之间的数据同步。对于那些频繁改变的数据缓存，需要一个描述机制。

一个更好的方式是使用一个内存数据网格来替换本地缓存，如图 4-29 所示。

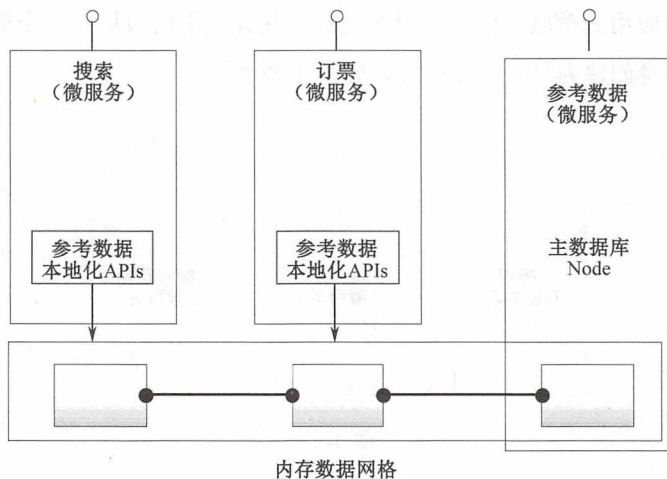


图 4-29

引用数据会被写入到数据网格，而嵌入到其他服务中的代理库只会有只读 API。这个减少了数据描述的需求，并且更加有效和一致。

用户接口和 Web 应用

在过渡阶段，我们需要同时保留老的和新的用户接口。有 3 种常用的方式应用于这个场景。

第 1 种方式是把老的和新的接口分开作为用户应用，在它们之间没有任何关联，如图 4-30 所示。

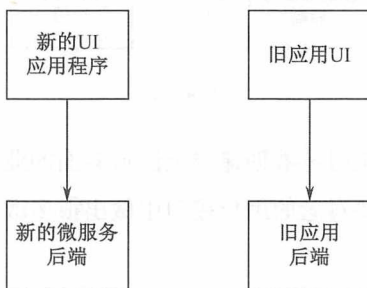


图 4-30

一个用户登录到新应用程序和旧应用程序中，就像是两个不同的应用，在它们之间没有单点登录（SSO）。这种方式很简单，并且没有开销。在多数情况下，这在业务上是不可接受的，除非它是针对两个不同的用户群体。

第 2 种方式是使用遗留用户接口作为主应用，然后当用户请求新应用的页面时把页面控制转移到新的用户接口，如图 4-31 所示。

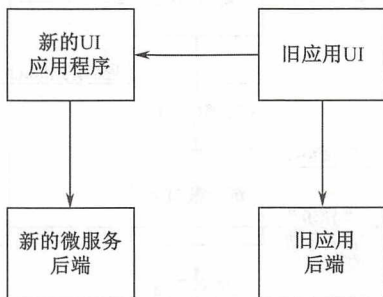


图 4-31

在这种情况下，由于新老应用都是运行在浏览器窗口中的基于 Web 的应用，用户可以得到无缝的体验。SSO 需要在老的和新的用户接口之间实现。

第 3 种方式是把现有遗留的用户接口直接集成到新的微服务后端，如图 4-32 所示。

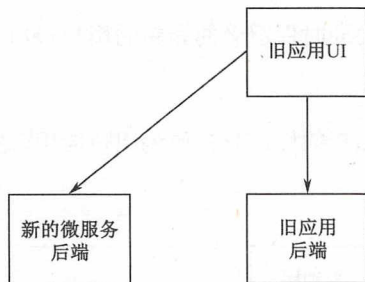


图 4-32

在这种情况下，新的微服务被创建成无图形界面的业务应用或服务。这个可能具有挑战性，因为可能需要对老的用户接口中做出很多改变，如引入服务调用、数据模型的转换等。

后面两种方式的另一个问题是，如何处理资源和服务的鉴权。

会话处理和安全

假设新服务是基于带有基于 token 进行鉴权的 Spring Security 来编写的，然而老的应用是使用它的本地身份存储的定制身份验证。

图 4-33 展示了如何继承新、老两种服务。

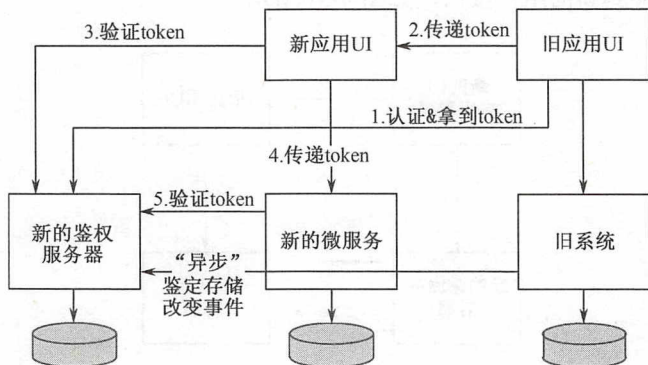


图 4-33

如图 4-33 所示,最简单的方法是使用鉴权服务创建一个新的身份存储,利用 Spring Security 把它作为一个新的微服务。这个会用于对我们未来所有微服务的资源和服务进行保护。

现有的用户接口应用鉴权本身不利于新的鉴权服务和获取 token。这个 token 会被发送到新的用户接口和微服务。这两种情况下,用户接口或者微服务都会向鉴权服务发送请求来验证给定的 token。如果 token 是合法的,那么 UI 或者微服务就会接受这个请求。

这里需要关注的是,老的身份识别存储需要跟新的进行同步。

测试策略

站在测试的角度,一个需要回答的很重要的问题是,如何确保所有功能都能像进行迁移之前那样运转?

所有需要进行迁移的服务在进行迁移或者重构之前都应该书写对应的集成测试用例。这可以确保一旦迁移完毕,我们可以得到同样预期的结果,并且系统的行为可以保持一致。需要准备好自动化回归测试包,并且在我们每次对新老系统进行更改时,都应该执行回归测试。

在图 4-34 中,对于每一个服务,我们需要针对 EJB 端点和微服务点分别进行一个测试。

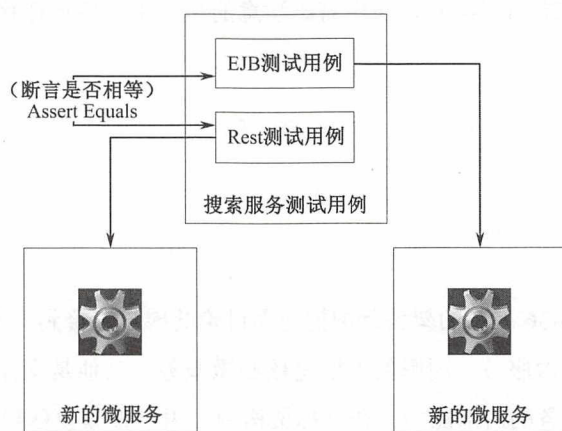


图 4-34

创建生态系统能力

在我们着手实际的迁移之前，需要创建在功能模型中提到的所有微服务的能力，这个能力模型我们曾在第 3 章“微服务概念的应用”中进行过阐述。这些是开发基于微服务的系统的先决条件。

除了这些能力，特定应用的功能也需要提前创建，如参考数据、安全和 SSO、客户和通知，一个数据仓库或者数据网格同样是一个必要的先决条件，一个有效率的方式是以渐进的方式创建这些能力，只有确实必要时才进行开发。

只有在需要时迁移模块

在第 3 章中，我们探讨过把单体应用转化为微服务的方法和步骤。有很重要的一点需要明白，没有必要把所有的模块都迁移到新的微服务架构，除非它确实需要。一个主要原因是迁移需要成本。

我们将回顾一些这样的场景。BrownField 已经决定使用一个外部收入管理系统来替代 PSS 收入管理功能，并且正在集中它们的账单功能，所以没有必要从老的系统中把账单功能迁移出来。从业务的角度看，迁移 CRM 并没有带来太多的价值。因此决定把 CRM 还是保留在老的系统中。业务计划搬到一个基于 saas 的 CRM 解决方案，作为它们云策略的一部分。同样需要注意的是，中途停止迁移会严重影响系统的复杂性。

目标架构

图 4-35 和图 4-36 展示的架构蓝图把前面讨论的内容整合到一个架构视图。图中的每一块代表一个微服务。阴影的方框是核心微服务，其他是支撑性微服务。图中还展示了每个微服务的内部能力。在目标架构中，用户管理被移到安全下面：



图 4-35

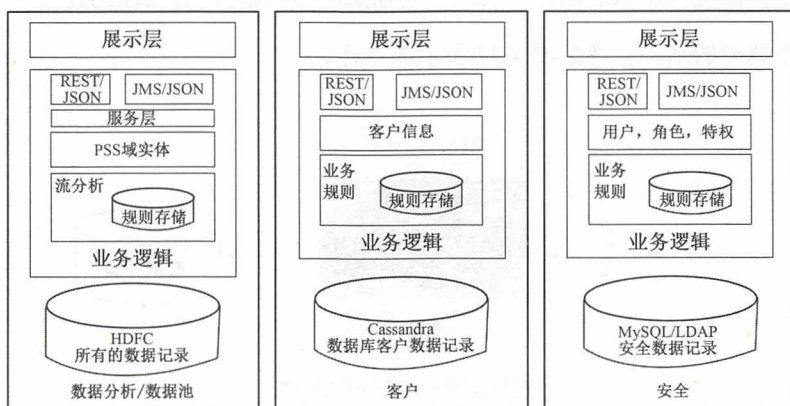


图 4-36

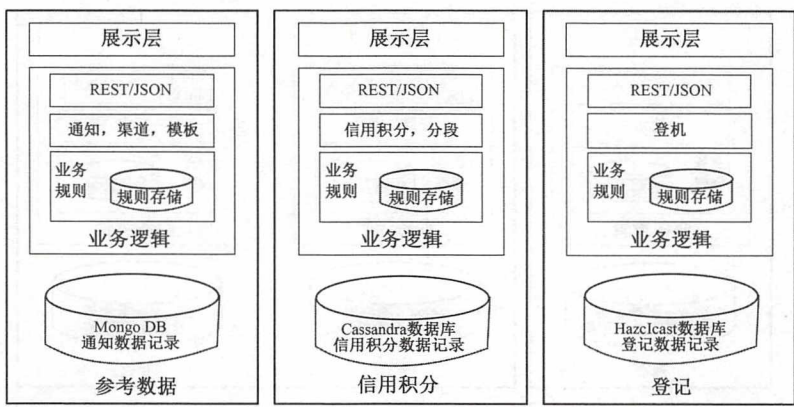


图 4-36 (续)

每个服务都有它自己的架构，通常由一个表现层、一个或多个服务端点、业务逻辑及数据库组成。正如我们所见，对于每个微服务，我们选择适合它们的不同数据库。它们中的每一个都是具有最小编排的自治服务。这些服务中的大部分使用服务端点与其他服务进行交互。

微服务的内部分层

在这一节中，我们会进一步探索微服务的内部结构。对于微服务的内部结构，没有一个标准可以用来遵循。相关经验是抽象简单微服务点背后的实现。

一个典型的结构会像是图 4-37 所示的这样。

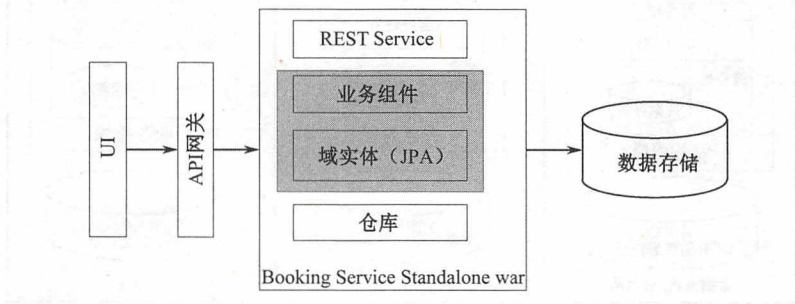


图 4-37

UI 通过一个服务网关访问 REST 服务。一个 API 网关可能对应一个微服务，也有可能对应多个微服务——这取决于我们想要 API 网关做什么。微服务点可能暴露

一个或者多个 REST 端点。这些端点反过来连接服务内部的一个业务组件。业务组件接着在域实体的帮助下执行所有的业务功能。一个仓库组件被用于和后台数据存储进行交互。

编制微服务

预订业务的逻辑和规则的执行存在于 Booking 服务的内部。重心依然存在于 Booking 服务中，通过一个或多个预订业务组件的形式。在内部，业务组件编排其他业务组件甚至外部服务暴露出来的私有 API。

如图 4-38 所示，Booking 服务内部调用它自己的组件来更新库存，而不是调用 Fare 服务。

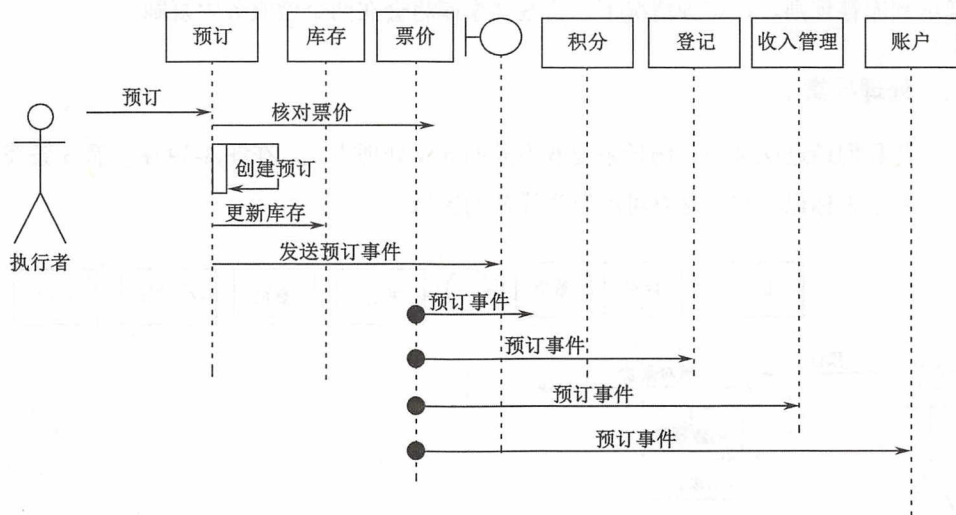


图 4-38

对于这个活动是否需要流程引擎？这个依赖于需求。在复杂的场景中，我们可能需要并行处理一系列的事情。例如，创建一个预订事件，内部应用一系列预订规则，它检验票价并且在预订之前检验是否有足够的库存。我们可能想要同时处理这些事情，在这种情况下，我们可能会使用 Java concurrency API 或者 reactive Java 库。

在极其复杂的情况下，我们可能会选择一个集成框架，如 Spring 或者在嵌入式模式下选择 Apache Camel。

与其他系统集成

在微服务中，我们使用 API 网关或者一个可靠的消息总线来集成其他的非微服务。

假设在 BrownField 中有另一个系统需要预订的相关数据。遗憾的是，这个系统不能监听 Booking 微服务发布的预订事件。在这种情况下，一个企业应用集成 (EAI) 可以作为一种解决方案，它监听我们的预订事件，然后使用一个本地适配器来更新数据库。

管理共享库

某个业务逻辑在不止一个微服务中使用。例如，Search 服务和 Reservation 服务都用到库存规则。在这种情况下，这些共享库将会在两个微服务中复制。

处理异常

让我们通过探讨预订场景来理解不同的异常处理方式。在图 4-39 中，有 3 条线被打上了叉标记。它们是有可能发生异常的区域。

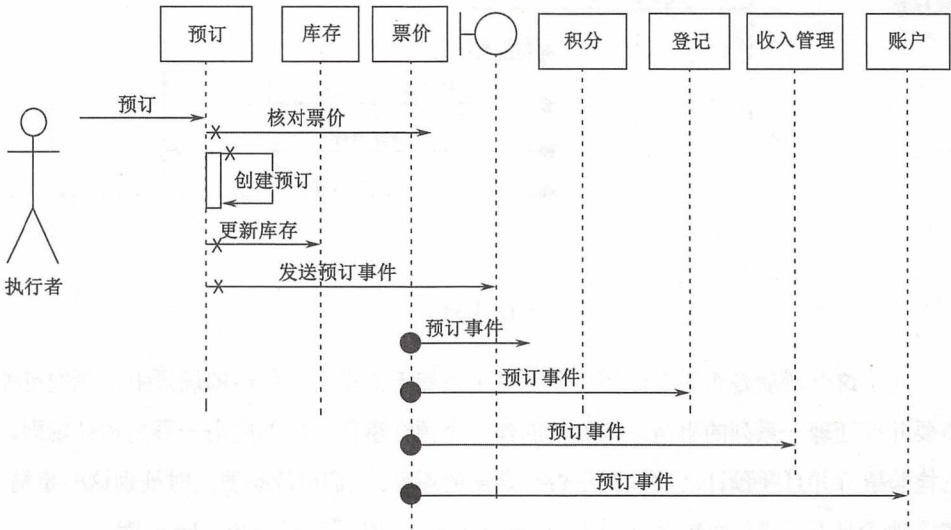


图 4-39

Booking 服务和 Fare 服务间使用的是同步通信方式。如果 Fare 服务不可用会发生

什么？单纯地向调用方抛出一个异常会导致收益减少。另一种想法是信任作为传入请求一部分的票价。当我们提供搜索服务时，搜索结果也会带有票价。当用户选择一个航班然后提交，请求中就会带有所选的票价。为了防止 Fare 服务不可用，我们信任输入的请求，并且接受预订。我们会使用一个熔断器和一个回滚服务，它们简单地创建带有一个特殊状态的订单，并且把订单入队列用于人工操作或者系统重试。

如果创建预订失败会怎样？如果创建一个预订意外失败，一个更好的方式是给用户抛回一个消息。我们可以尝试另一种方式，但是这会给系统增加整体复杂性。这同样适用于库存更新。

在创建预订和更新库存的场景中，我们应避免预订创建成功，但是库存更新失败的情况。因为库存很关键，应该让创建预订和更新库存存在一个本地事务中。由于两个组件存在于同一个子系统下，这种方式是可行的。

如果我们考虑 Check-in 服务的场景，Check-in 服务发送一个事件到 Boarding 服务和 Booking 服务，如图 4-40 所示。

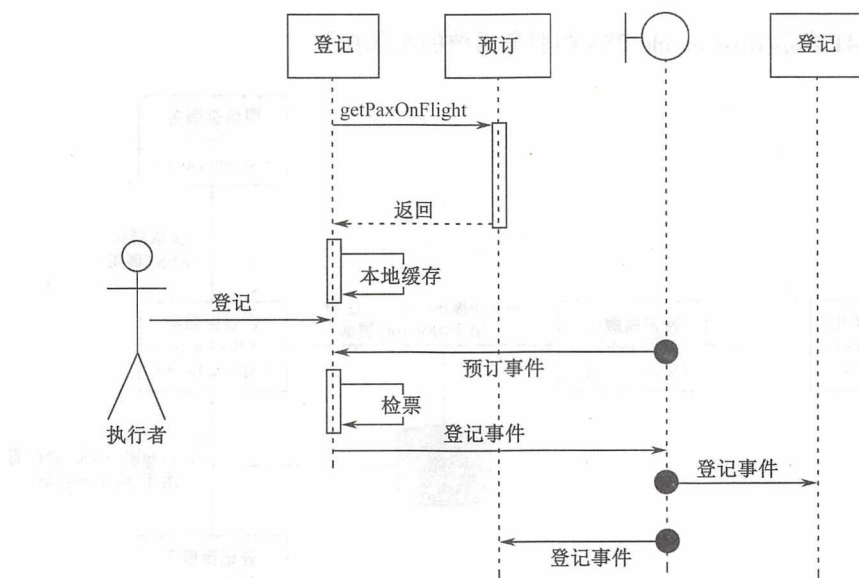


图 4-40

考虑这样一个场景，Check-in 服务在 Check-in 事件发出的瞬间不可用。其他消费者处理了这个事件，但是实际的 Check-in 被回滚。这是因为我们没有使用一个两

阶段提交。在这种情况下，我们需要一个机制来恢复这个事件。这个可以通过捕获异常，然后发送另一个 Check-in 取消事件来实现。需要注意的是，为了最少地使用修正事务，把发送 Check-in 事件移到 Check-in 事务的最后。这样做减少了发送事件之后失败的概率。

另外，如果 Check-in 成功但是发送 Check-in 事件失败会怎样？我们可以考虑两种方法。第一种是调用一个回退服务来在本地存储这个事件，然后在之后使用另一个扫描清除程序来发送这个事件。还可以多次重试发送这个事件，这个会增加更多的复杂性，并且效率不高。另一种方式是把异常抛回给调用方，以便调用方可以重试，但是，这种方式在客户交互角度看并不总是太好。第一种方式对系统的健壮性更好，我们需要根据业务来进行权衡，找到最好的一种方案。

目标实现视图

图 4-41 表示 BrownField PSS 微服务系统的实现视图。

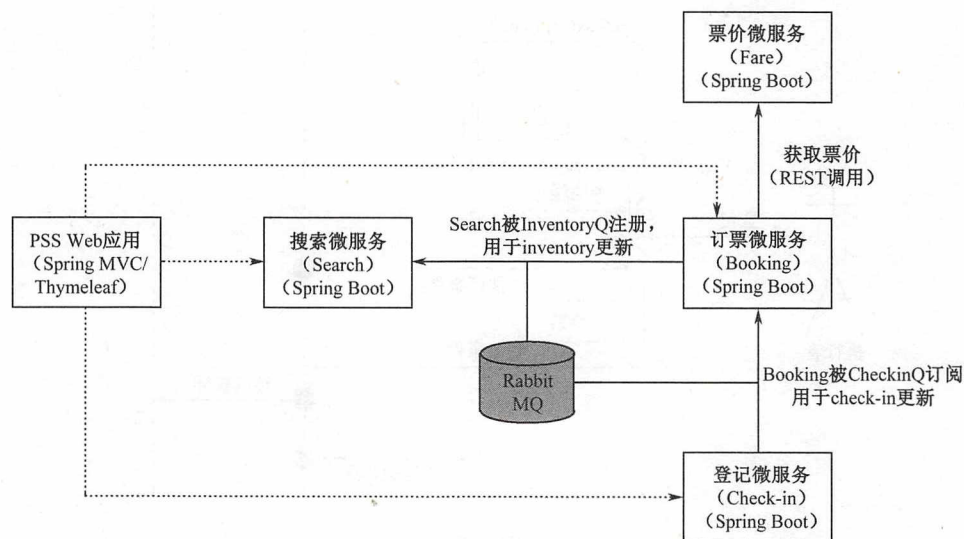


图 4-41

如图 4-41 所示，我们实现了 4 个微服务作为例子：Search、Fare、Booking 及

Check-in 服务。为了测试应用，还有一个使用 Spring MVC 和 Thymeleaf 模板开发的网站应用。通过 RabbitMQ 实现了异步消息机制。在这个样例实现中，为了演示的目的，默认的 H2 数据库用于内存存储。

这一节中的代码演示了在本章“回顾微服务能力模型”一节中强调的所有能力。

实现项目

如下面表格所总结的那样，BrownField 航空的 PSS 微服务系统的基本实现中有 5 个核心项目。表格还展示了这些项目所使用的端口范围，以保证在全书中保持一致性。

微服务	项目	端口范围
Book 微服务	chapter4.book	8060-8069
Check-in 微服务	chapter4.checkin	8070-8079
Fare 微服务	chapter4.fares	8080-8089
Search 微服务	chapter4.search	8090-8099
Website	chapter4.website	8001

Website 是用来测试 PSS 微服务的 UI 应用程序。

本例中所有的微服务项目都遵循图 4-42 展示的包结构。

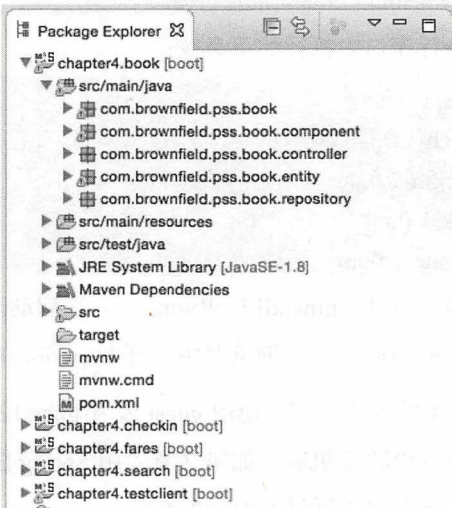


图 4-42

不同的包及对应的用途如下：

- 根目录 (com.brownfield.pss.book) 包含默认的 Spring Boot 应用。
- component 包包含了所有服务组件，它们是业务逻辑实现的地方。
- controller 包包含了 REST 端点和消息端点。Controller 类内部利用组件类进行执行。
- entity 包包含 JPA 实体类来映射到数据库表。
- Repository 类存在于 repository 包下，基于 Spring Data JPA。

运行和测试项目

遵循下面列出的步骤来构建和测试本章的微服务例子：

(1) 使用 Maven 分别构建各个项目。需要确保 test 标记是关闭的。测试程序会假设其他依赖的服务都是开启并运行的，如果所依赖的服务不可用，会导致失败。在我们的例子中，Booking 和 Fare 服务有直接的依赖。我们会在第 7 章“日志记录和监控微服务”中学习如何绕过这个依赖：

```
mvn -Dmaven.test.skip=true install
```

(2) 运行 RabbitMQ 服务：

```
rabbitmq_server-3.5.6/sbin$ ./rabbitmq-server
```

(3) 在另一个命令行窗口中执行如下命令：

```
java -jar target/fares-1.0.jar  
java -jar target/search-1.0.jar  
java -jar target/checkin-1.0.jar  
java -jar target/book-1.0.jar  
java -jar target/website-1.0.jar
```

(4) Website 项目有一个 CommandLineRunner，它在启动时执行了所有的测试用例。一旦所有的服务都成功启动，在浏览器中打开 <http://localhost:8001>。

(5) 浏览器需要基本的安全证书，使用 guest 或者 guest123 作为证书。本例中只显示该网站基本的安全身份验证机制。如第 2 章“用 Spring Boot 构建微服务”所提到的那样，服务级别的安全可以通过 OAuth2 实现。

(6) 输入正确的安全证书会显示图 4-43 的界面，这是我们 BrownField 应用的主页面。

BrownField Airline Search CheckIn

Flight Search

traveling from NYC

going to SFO

planning on 22-JAN-16

SUBMIT

图 4-43

(7) SUBMIT 按钮调用 Search 微服务来获取符合搜索条件的可用航班。一些航班会在 Search 微服务启动时提前生成。如有需要，编辑 Search 微服务的代码来生成其他的航班。

(8) 图 4-44 展示了输出的航班列表。Book 链接会把我们引入所选择航班的预订界面。

BrownField Airline Search CheckIn

Available Flights

#	Flight	From	To	Date	Fare	
2	BF101	NYC	SFO	22-JAN-16	101	Book
3	BF105	NYC	SFO	22-JAN-16	105	Book
4	BF106	NYC	SFO	22-JAN-16	106	Book

图 4-44

(9) 图 4-45 展示了预订界面。用户可以输入乘客详情，然后通过单击 CONFIRM 按钮创建一条订单。在这里会调用 Booking 微服务，在其内部会调用 Fare 微服务。它还会向 Search 微服务返回一条消息。

BrownField Airline Search CheckIn

Selected Flight

BF101 NYC SFO 22-JAN-16 101

First Name Rajesh

Last Name RV

Gender Male

CONFIRM

图 4-45

(10) 如果成功预订，接下来的确认界面会带有一条预订参考号码，如图 4-46 所示。

(11) 接下来测试 Check-in 微服务。我们单击屏幕上方菜单栏的 CheckIn 按钮，使用前面得到的预订参考号码来测试 Check-in 微服务，图 4-47 展示了这一过程。

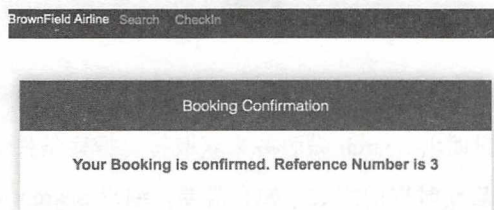


图 4-46

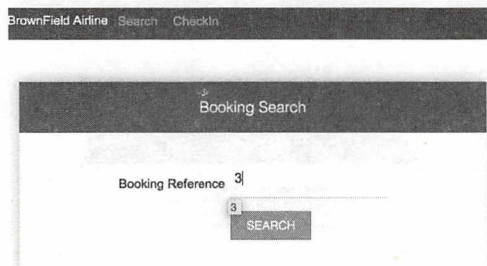


图 4-47

(12) 单击上图中的 SEARCH 按钮调用 Booking 微服务，然后得到预订信息。单击 CheckIn 链接来执行 Check-in。这会调用 Check-in 微服务，如图 4-48 所示。

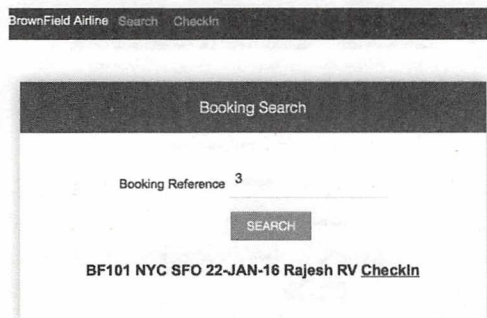


图 4-48

(13) 如图 4-49 所示,一旦成功登记,会展示带有一个确认编号的确认信息。这是通过在内部调用 Check-in 服务实现的。Check-in 服务发送一条消息到 Booking 服务来更新登记状态。

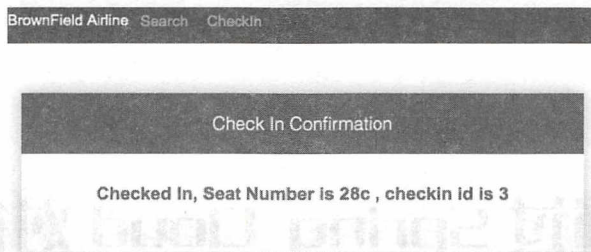


图 4-49

总结

在本章中,我们使用 Spring Boot 的基本功能实现并测试了 BrownField PSS 微服务。我们学习了如何使用微服务架构来解决一个实际的用例。

我们分析了在实际过程中,把一个项目从单个应用向微服务进行演化过程中的各个阶段。我们还评估了多种方法的利弊,以及在迁移过程中所遇到的阻碍。最后,对我们探讨的用例进行了端到端微服务设计。对一个成熟的微服务实现所进行的设计与实现同样得到了验证。

在第 5 章,我们会看到 Spring Cloud 项目是如何将我们开发的 BrownField PSS 微服务部署到互联网上。

第 5 章

通过 Spring Cloud 对微服务 进行扩（缩）容



想要高效管理互联网级别的微服务，仅仅依靠 Spring Boot 的框架是不够的。Spring Cloud 提供了一系列专用组件来实现这些附加功能。

本章将详细介绍 Spring Cloud 中的各种组件，包括 Eureka、Zuul、Ribbon 和 SpringConfig。这些组件都是用来支撑第 3 章“微服务概念的应用”中建立的微服务功能模型。并且我们将在本章中着重介绍 Spring Cloud 组件如何帮助我们对 BrownField 航空的 PSS 微服务系统进行扩容和缩容。

阅读本章后，您会学到下列知识：

- SpringConfig Server 如何获取外部配置。
- Eureka 服务器配置服务注册与发现。
- Zuul 关联服务代理与网关。
- 自动化微服务注册和服务发现实现。

- Spring Cloud 消息实现异步微服务。

回顾微服务

在第 3 章“微服务概念的应用”的微服务功能模型例子的基础上，本章中的例子会深入研究下列微服务功能，如图 5-1 高亮显示。



图 5-1

- 负载均衡器。
- 服务注册。
- 配置服务。
- 可靠的 Cloud 消息。
- API 网关。

回顾 BrownField 航空的 PSS 系统实践

在第 4 章“微服务的演变——一个案例的学习”中，我们使用 Spring 框架和 Spring Boot 为 BrownField 航空设计了一套基于微服务的 PSS 系统。这个系统能满足小并发请求的交易。然而，对于部署大规模（成百上千个微服务）的企业级应用，这还远远不够。

在第 4 章中，我们研发了 4 个微服务：搜索、预订、询价、登记。我们也研发了一个测试这些微服务的网站。

目前为止，在我们的微服务实践过程中，实现了以下功能：

- 为了实现业务功能，每一个微服务都会暴露一系列 REST/JSON 接口。
- 每一个微服务都会通过 Spring 框架实现特定的业务功能。
- 每一个微服务都使用 H2（一种内存数据库）存储它自己的数据。
- 使用 Spring Boot 搭建微服务，内置的 Tomcat 服务器可以用作 HTTP 监听。
- RabbitMQ 提供外部消息通信服务，搜索、预订、签到这些微服务通过异步消息彼此协作。
- Swagger 与所有微服务集成，负责记录 REST API。
- 开发了一个基于 OAuth2 的安全机制来保护微服务系统。

什么是 Spring Cloud

Spring Cloud 项目是 Spring 团队的一个大型项目，提供了一系列易用的 Java Spring 库，实现了分布式系统需要的一系列通用的模式。不要被 Spring Cloud 这个名字迷惑，它并不是一个云解决方案。确切地说，当您需要开发遵循十二因素应用原则的针对云部署的应用时，Spring Cloud 提供了大量的必要功能。在 Spring Cloud 的帮助下，开发者们只需要专注于使用 Spring Boot 来开发业务功能，就可以使系统具备分布式、容错和自我恢复功能。

Spring Cloud 的解决方案不用考虑部署环境，您可以在个人电脑或者弹性云上开发部署。使用 Spring Cloud 开发的云部署解决方案也不挑剔云服务的提供商（Cloud Foundry、AWS、Heroku 等都可以）。在没有 Spring Cloud 的岁月里，开发者们无法使用云计算供应商提供的原生服务，而且会造成与 Paas 提供者之间的深度耦合。开发者只能被迫去写大量的样板代码来构建这些服务。Spring Cloud 也提供了简单、易于使用、对 Spring 友好的 API。这些 API 抽象自底层云提供者的复杂服务 API（如 AWS 通知服务的 API），降低了开发者的使用难度。

根据 Spring “惯例优于配置”的原则，Spring Cloud 的所有配置都是默认的，能够帮助开发者快速上手。Spring Cloud 化繁为简，提供了简单的声明式配置。拥有更小覆盖范围的 Spring Cloud 组件对开发者非常友好，也更易于开发本地云应用。

Spring Cloud 为开发者的不同需求提供了许多可选的解决方案。例如，想要实现服务注册，可以选用 Eureka、ZooKeeper 或者 Consul。Spring Cloud 的组件之间耦合性很低，开发者可以根据自己的实际使用需求去选择采用某个组件。

Spring Cloud 和 Cloud Foundry 的区别？

Spring Cloud 是开发互联网范围的 Spring Boot 应用时的开发配套元件，Cloud Foundry 则是一个开源平台，可以构建、部署和扩（缩）容应用。

Spring Cloud

Spring Cloud 项目是一个囊括了众多组件的 Spring 项目。这些组件的版本在名为 spring-cloud-starter-parent 的 BOM 文件中申明。

本书中，我们依赖的是 Spring Cloud 的 Brixton.RELEASE 版本。

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifactId>
<version>Brixton.RELEASE</version>
<type>pom</type>
<scope>import</scope>
</dependency>
```

spring-cloud-starter-parent 文件标明了子组件的不同版本，例如：


```
<spring-cloud-aws.version>1.1.0.RELEASE</spring-cloud-aws.version>
<spring-cloud-bus.version>1.1.0.RELEASE</spring-cloud-bus.version>
<spring-cloud-cloudfoundry.version>1.0.0.RELEASE</spring-cloudcloudfoundry.version>
<spring-cloud-commons.version>1.1.0.RELEASE</spring-cloud-commons.version>
<spring-cloud-config.version>1.1.0.RELEASE</spring-cloud-config.version>
<spring-cloud-netflix.version>1.1.0.RELEASE</spring-cloud-netflix.version>
<spring-cloud-security.version>1.1.0.RELEASE</spring-cloud-security.version>
<spring-cloud-cluster.version>1.0.0.RELEASE</spring-cloud-cluster.version>
<spring-cloud-consul.version>1.0.0.RELEASE</spring-cloud-consul.version>
<spring-cloud-sleuth.version>1.0.0.RELEASE</spring-cloud-sleuth.version>
<spring-cloud-stream.version>1.0.0.RELEASE</spring-cloud-stream.version>
<spring-cloud-zookeeper.version>1.0.0.RELEASE </spring-cloudzookeeper.version>
```

Spring Cloud 的版本名字采用了伦敦地铁站的名字，根据字母表的顺序来对应版本时间顺序。例如，最早的 Release 版本是 Angel，第二个 Release 版本是 Brixton。

Spring Cloud 组件

每个 Spring Cloud 组件都专门针对某些分布式系统功能。图 5-2 中灰底框是 Spring Cloud 的功能，上面及下面的白底框是 Spring Cloud 中对应实现这些功能的子项目。



图 5-2

Spring Cloud 有如下功能：

- 分布式配置：开发、测试和生产等不同的环境中运行多个微服务实例，配置属性将会很难管理。因此，有一个配置中心去管理这些配置属性是非常重要的。分布式配置管理模块可以中心化和具体化微服务配置参数。Spring Cloud 配置使用 git 或者 svn 作为版本管理工具，Spring Cloud Bus 支持向多个订阅者更新配置文件，通常是针对一个微服务实例。另外，ZooKeeper 或者 HashiCorp 的 Consul 也可以用作分布式配置文件管理。
- 路由：路由是一个 API 网关组件，主要用于反向代理转发消费者发送请求到服务提供接口。我们也可以用这个网关组件实现基于软件的路由和过滤。Zuul 是一个轻量级网关解决方案，可以在流量整形和请求/响应转换中实现细粒度的控制。
- 负载均衡：负载均衡功能要求实现一个基于软件的负载均衡器模块，该模块可以利用负载均衡算法将请求路由给可用的服务器。Ribbon 就是 Spring Cloud 中一个具有这样功能的子项目。Ribbon 可以作为独立组件使用，也可以在分流路由中和 Zuul 无缝协作。
- 服务注册和发现：服务注册和发现模块的作用是，当一个服务可用并且能够接受通信流的时候，使其注册到一个注册中心，通知它们的存在且可以被发现。消费者在注册中心就可以知道哪些服务可用并且得知其终端位置。在很多情况下，没有约束好的注册中心差不多是个垃圾堆。但是有了注册中心组件的存在，让整个生态系统变得智能。Spring Cloud 下面有许多支持注册和发现的子系统。例如，Eureka、ZooKeeper 和 Consul 是 3 个具有注册功能的子项目。
- 服务到服务的请求：Spring Cloud Feign 子项目提供了声明式的方法，可以同步实现 RESTful 风格的服务到服务的请求。声明式的方法允许应用与 POJO 接口而不是低级的 HTTP 客户端 API 协同工作。本质上 Feign 在通信中使用响应式库。
- 熔断机制：这个子项目使用了保险丝熔断的设计模式。当主服务失败的时候，系统就会把流量切到另一个临时的“保底”服务上去，从而切断回路。当主服务恢复正常，它也可以自动切回到主服务上去，最终它提供一个监

控服务状态变化的仪表盘。Spring Cloud Hystrix 实现了熔断机制，Hystrix Dashboard 则可以可视化监控服务。

- 全局锁、内部选举和集群状态：当进行大规模部署的时候，集群管理和协作中也需要这个功能。它也可以在诸如序列生成等情况下提供全局锁。Spring Cloud 集群可以通过 Redis、ZooKeeper 和 Consul 等实现这些功能。
- 安全：对于使用如 OAuth2 等具体授权提供者的本地云分布式系统，安全功能是必备的。Spring Cloud 安全项目使用可定制的授权和资源服务器来实现这一功能。它也提供许多微服务必备的 SSO 功能。
- 大数据支持：当您需要大数据解决方案中的数据服务和数据流时，大数据支持功能是离不开的。Spring Cloud Stream 和 Spring Cloud Data Flow 项目实现了这些大数据功能。Spring Cloud Data Flow 是基于 Spring XD 重新设计的版本。
- 分布式追踪：分布式追踪能力帮助我们在多个微服务实例之间实现线程和关联服务的转换。Spring Cloud Sleuth 通过在通常的分布式追踪数据模型上提供一个抽象模型来实现分布式微服务的追踪方案（如 64 位 ID 支持下的 Zipkin 和 HTrace）。
- 分布式消息：Spring Cloud Stream 在如 Kafka、Redis 和 RabbitMQ 等可靠消息解决方案中提供了消息整合。
- 云支持：Spring Cloud 提供一系列支持多个连接器、整合机制以及顶层抽象出不同的云提供商的功能，如 Cloud Foundry 和 AWS。

Spring Cloud 和 Netflix OSS

微服务部署中，许多极其重要的 Spring Cloud 组件来自 Netflix 开源软件（Netflix OSS）中心。Netflix 是微服务领域的先驱者。为了管理大规模的微服务，Netflix 的工程师们原创了很多工具。这些工具填补了 AWS 平台管理 Netflix 服务的一些软件空白。后来，Netflix 开源了这些组件，并且让它们在 Netflix 平台下可以开放使用。这些组件在生产系统中广泛使用，并且在 Netflix 的大规模微服务部署中经历了实战的考验。

Spring Cloud 为这些 Netflix 组件提供了高度抽象化的概念，让它们对 Spring 开

发者更友好。Spring Cloud 也提供了声明机制，并能够和 Spring Boot 和 Spring 框架高度协同工作。

建立 BrownField PSS 的环境

在本章中，我们将会使用 Spring Cloud 功能来改进第 4 章中开发的 BrownField PSS 微服务。我们也会检验如何使用 Spring Cloud 组件来使这些服务达到企业级程度。

本章中随后的几节将会研究如何使用 Spring Cloud 项目提供的功能，来对云范围部署的微服务进行扩（缩）容。后面的部分，我们会详细探讨 Spring Cloud 的功能，如用 Spring Config 服务器，基于 Ribbon 的服务负载均衡，Eureka 发现服务，网关中 Zuul 的应用，以及基于信息服务交互的 Spring Cloud 信息发送。我们会着重介绍第 4 章中的 BrownField PSS 微服务改善后的功能。

为了准备本章的环境，将之前的项目代码（第 4、5 章）导入并且重命名为一个新的 STS 工作空间。

源代码可以从第 5 章的代码文件中找到。

Spring Cloud Config

Spring Cloud Config 服务器是一个统一的管理分布式系统的配置中心。通过其中的应用和服务，可以部署、访问和管理所有的运行时配置属性项。Spring Config 服务器也支持配置属性的版本控制。

在 Spring Boot 之前的例子中，所有的配置参数都是从项目中的配置文件（application.Properties 或者 application.yaml）中读取的，这种方式将所有属性放在独立于代码之外的文件，是很好的方式。美中不足的是，当微服务从一个环境迁移到另一个环境中，这些属性也需要修改，也就是说，应用需要重新构建。这违反了十二因素的一点——一次构建，可以在不同的环境之间运行。

更好的途径是使用配置的概念。正如我们在第 2 章中介绍过的，配置可以根据

不同的环境适配不同的属性。配置文件配置在 `application-{profile}.properties` 文件中。例如，`application-development.properties` 针对开发环境。

然而，这种方式的不足之处在于，应用中的配置是静态打包的。任何配置属性的改变，都会需要整个应用重新构建。

替代方式是，将配置属性移到应用部署包的外部。配置属性也可以有很多方式从外部文件中获取。

- 用 JNDI 命名空间（`java: comp/env`）从外部 JNDI 服务器获取。
- 用 `java` 系统文件（`system.Properties`）或者使用 `-D` 命令行选项。
- 使用 `PropertySource` 配置：

```
@PropertySource("file:${CONF_DIR}/application.properties")
public class ApplicationConfig {
}
```

- 使用命令行参数让文件指向外部地址：

```
java -jar myproject.jar --spring.config.location=
```

JNDI 方式代价很高而且缺乏灵活性，复制的时候很难，而且没有版本控制。`System.Properties` 在大规模部署中不够灵活。最后两种方式则依赖于服务器中本地或共享的系统文件。

对于大规模部署，我们需要一个简单并且足够强大的中心化配置管理解决方案。

如图 5-3 所示，所有的微服务通过指向一个中心服务器来得到需要的配置参数。然后微服务会在本地缓存这些参数来提高性能。一旦配置状态改变，`Config` 服务器会通知所有订阅的微服务去更新本地缓存。`Config` 服务器也会使用配置来根据环境提供特定的值。

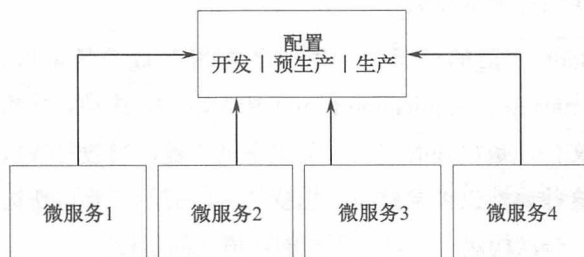


图 5-3

在图 5-4 中，为了构建配置服务器，在 Spring Cloud 项目中有许多可选的方案，如 Config 服务器、ZooKeeper 配置、Consul 配置。然而，这一章只会着重介绍 Spring Config 服务器的实现。



图 5-4

Spring Config 将属性值存储在一个有版本控制的仓库中，如 Git 或 SVN。Git 库可以是本地或者远程的。在大规模分布式微服务部署中，一个高可用的远程 Git 服务器弥足珍贵。

Spring Cloud Config 服务器的结构如图 5-5 所示。

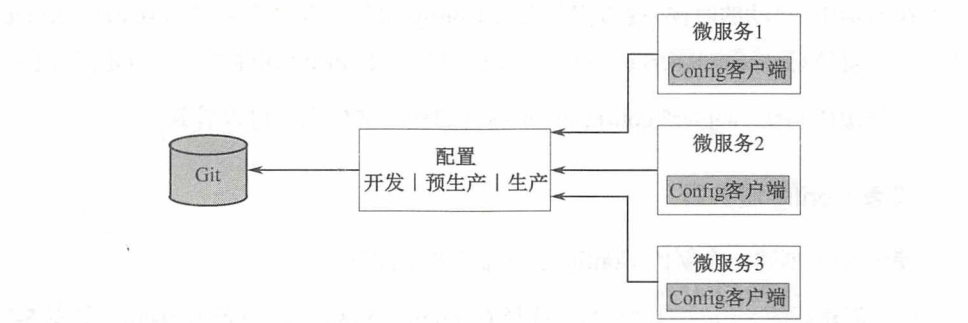


图 5-5

嵌入在 Spring Boot 中的 Config 客户端使用简单的声明机制从中心化配置服务器中找到所需配置，并且将属性存储到 Spring 环境中。配置属性可能是应用层级的配置（如每日贸易限制），或者基础设施相关的配置（如服务器 URL、资格证书等）。

与 Spring Boot 不同，Spring Cloud 使用了一个 Bootstrap context，而这个语境是主应用的父类语境。Bootstrap context 负责从 Config 服务器的 bootstrap.yaml 或者

bootstrap.properties 文件加载配置属性,为了能让这一机制在 Spring Boot 应用中生效,需要将 application.*文件重命名为 bootstrap.*。

接下来呢

接下来的几部分将会介绍如何在现实场景中使用 Config 服务器。我们会修改之前的搜索微服务,让它借助 Config 服务器的魔法,图 5-6 展示了整个流程。

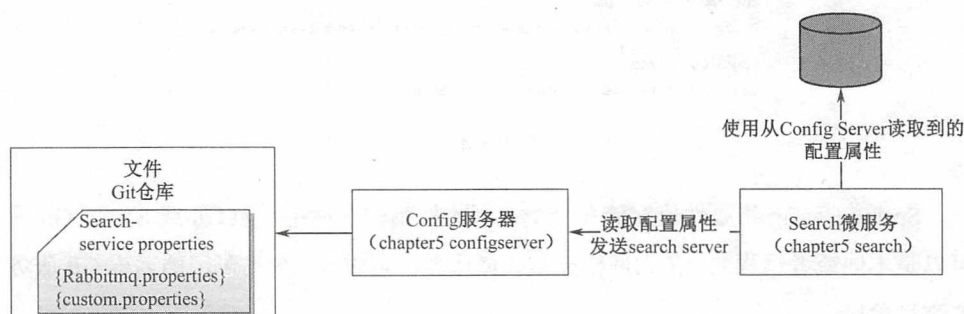


图 5-6

在本例中,启动的时候,搜索服务会向 Config 服务器发送服务名为 search-service 并请求配置信息, search-service 的配置属性包括 RabbitMQ 属性和一个自定义属性。

完整源代码在 chapter5.configserver 项目的代码文件夹中可以看到。

设置 Config 服务器

使用 STS 创建一个新的 Config 服务器需要下面几步:

(1) 创建新的 Spring 启动项目,选择 Config 服务器和 ActuatorActuator,如图 5-7 所示。

(2) 创建 Git 库。将项目指向一个远程 Git 配置仓库,如 <https://github.com/spring-cloud-samples/config-repo>。这个链接只是一个例子,实际创建的时候,我们需要使用自己的 Git 仓库。

(3) 一个替代方案是,使用基于本地文件系统的 Git 仓库。在现实生产场景中,应该有一个外部 Git。本章中的 Config 服务器就会使用基于本地文件系统的 Git 仓库作为范例。

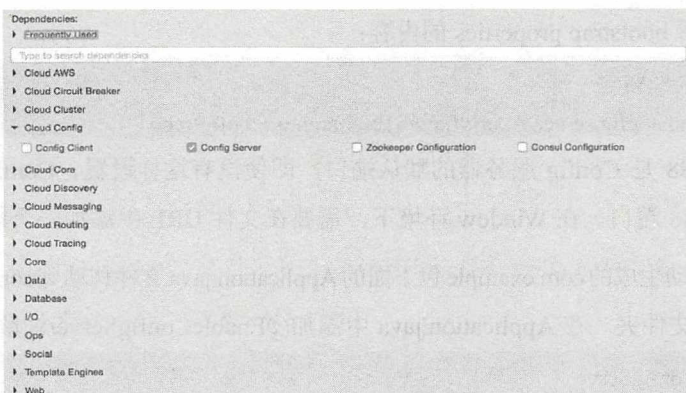


图 5-7

(4) 输入下面的命令来创建一个本地的 Git 仓库：

```
$ cd $HOME
$ mkdir config-repo
$ cd config-repo
$ git init
$ echo message : helloworld > application.properties
$ git add -A
$ git commit -m "Added sample application.properties"
```

通过以上命令，一个名为 application.properties 的属性文件就被创建好了。除此之外，还有一个 message 属性和 helloworld 值被创建好。

application.properties 只是范例，我们会在后面的部分中修改它。

(5) 接下来，使用上一步中建立的 Git 库修改 Config 服务器中的配置。将 application.properties 重命名为 bootstrap.properties，如图 5-8 所示。

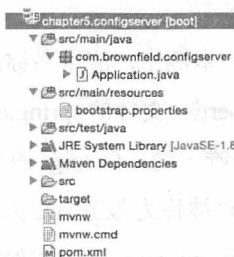


图 5-8

(6) 编辑 bootstrap.properties 的内容:

```
server.port=8888
spring.cloud.config.server.git.uri: file://${user.home}/Config-repo
```

端口 8888 是 Config 服务器的默认端口。即使没有这样设置, Config 服务器还是会绑定 8888 端口。在 Window 环境下, 需要在文件 URL 中添加一个额外的/。

(7) 将自动生成的 com.example 包下面的 Application.java 文件移动到 com.brownfield.configserver 文件夹。在 Application.java 中添加 @EnableConfigServer 注解。

```
@EnableConfigServer
@SpringBootApplication
public class ConfigserverApplication {
```

(8) 右键单击项目, 选择 “run as a Spring Boot app” 启动 Config 服务器。

(9) 访问 <http://localhost:8888/env> 可以查看服务器是否成功运行。如果一切顺利, 这个页面中将会列出所有环境配置。

(10) 访问 <http://localhost:8888/application/default/master> 可以查看 application.properties 中的具体属性。浏览器将会展示 application.properties 中类似于下面的配置属性。

```
{"name":"application","profiles":["default"],"label":"master","version":"6046fd2ff4fa09d3843767660d963866ffcc7d28","propertySources":[{"name":"file:///Users/rvllabs /config-repo /application.properties","source":{"message":"helloworld"}}]}
```

理解 Config 服务器的 URL

在之前的部分中, 我们用 <http://localhost:8888/application/default/master> 得到属性。我们如何理解这个 URL 呢?

URL 中的第一部分是应用名。本例中, 应用名称是 application。这个应用名是 Spring Boot 应用里的 bootstrap.properties 文件的 spring.application.name 属性赋予的逻辑名称。每个应用的名字都必须是唯一的。Config 服务器会从仓库中用这个名称解析和得到正确的属性。应用名有时也被称为服务 ID。如果有一个称为 myapp 的应用, 相应的在配置库中就应该有一个 myapp.properties 文件存储该应用所有的相关属性。

URL 的第二部分代表 profile。在应用的仓库中可以有一个或多个 profile。Profile

可以在不同的场景中使用。两个最常见的使用场景是，区分不同的环境（如 Dev、Test、Stage、Prod），或者区分服务器配置（如首要、次要）。前者代表应用的不同环境，后者代表应用部署的不同服务器。

Profile 名是逻辑名称，将会用来匹配仓库中的文件名。默认 profile 名称是 default。为了在不同的环境中配置这些属性，我们必须配置不同的文件，如下所示。在本例中，第一个文件用于研发环境，第二个用于生产环境。

```
application-development.properties
```

```
application-production.properties
```

分别对应下面的 URL：

```
http://localhost:8888/application/development
```

```
http://localhost:8888/application/production
```

URL 的最后一部分是标签，默认称为 master。它是可选的 Git 标签，以备不时之需。

简而言之，URL 符合下面的模式：

```
http://localhost:8888/{name}/{profile}/{label}
```

即便 URL 中不输入 profile，也可以找到配置。在前面的例子中，下面的 3 个 URL 都可以找到我们需要的配置信息：

```
http://localhost:8888/application/default
```

```
http://localhost:8888/application/master
```

```
http://localhost:8888/application/default/master
```

从客户端链接 Config 服务器

在前面的部分，Config 服务器是使用浏览器创建并且访问的。这一节中，搜索微服务会被改为使用 Config 服务器，也就是说，搜索微服务会作为 Config 客户端。

按照下面的步骤，可以避免从 application.properties 文件中读取属性，而是直接从 Config 服务器读取：

（1）添加对 Spring Cloud Config 的依赖，在 pom.xml 中加入 Actuator（原来没有 Actuator 的情况下）。想要刷新配置属性，Actuator 是必须的。

```
<dependency>
```

```
<groupId>org.springframework.cloud</groupId>
```

```
<artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

(2) 想要修改前面几章中的 Spring Boot 搜索微服务，就得在 Spring Cloud 中加入下列依赖。如果项目是从头创建的，这一步就可以跳过了。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Brixton.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

(3) 下面的截图是 Cloud 启动库的选择窗口。如果应用是从头开始构建的，按图 5-9 的方式选择。

(4) 将 Application.properties 重命名为 bootstrap.properties，然后添加一个应用名称和配置服务器 URL。如果 Config 服务器是运行在默认端口（8888）上的，配置服务器的 URL 就不是必须的了。

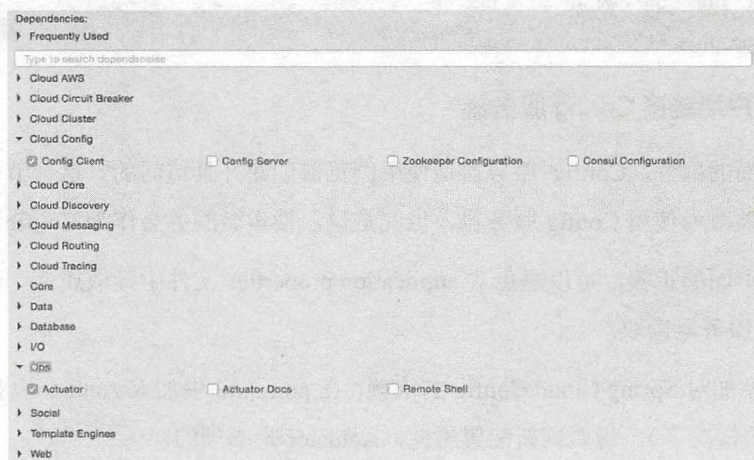


图 5-9

新的 bootstrap.properties 内容如下：

```
spring.application.name=search-service
spring.cloud.config.uri=http://localhost:8888
server.port=8090
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

search-service 是搜索微服务的逻辑名称，相当于它的服务 ID。Config 服务器会查找 search-service.properties 来解析这些属性。

(5) 为 search-service 创建一个新的配置文件。在 Git 库的 config-repo 文件夹下面建一个新的 search-service.properties。需要注意的是，search-service 是 bootstrap.properties 文件中搜索微服务的服务 ID。将特定服务的属性从 bootstrap.properties 移动到新的 search-service.properties 中。

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

(6) 为了实现配置属性的中心化及属性改变的通知，在属性文件中添加一个新的特定应用属性，添加 originairports.shutdown 以从搜索结果中暂时去掉一个机场。用户搜索关闭列表中的机场时，不会得到任何航班：

```
originairports.shutdown=SEA
```

本例中，搜索 SEA 为起点的时候，搜不到任何航班结果。

(7) 执行下列命令，把这个新文件提交到 Git 仓库。

```
git add -A .
git commit -m "adding new configuration"
```

(8) 最终的 search-service.properties 文件是这样的：

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

```
originairports.shutdown:SEA
```

(9) chapter5.search 项目中的 bootstrap.properties 如下:

```
spring.application.name=search-service
server.port=8090
spring.cloud.config.uri=http://localhost:8888
```

(10) 使用配置参数 originairports.shutdown 修改搜索微服务代码。使用 Refresh Scope 注解类级, 从而当属性发生改变的时候可以刷新。在这种情况下, 给 SearchRestController class 添加刷新域:

```
@RefreshScope
```

(11) 刚刚我们在 Config 服务器下面添加了新的属性, 现在添加实例变量作为它的占位符。search-service.properties 中的属性名称必须符合:

```
@Value("${originairports.shutdown}")
private String originAirportShutdownList;
```

(12) 如下所示, 修改应用代码(搜索方法)来使用这个属性:

```
@RequestMapping(value="/get", method =
RequestMethod.POST)
List<Flight> search(@RequestBody SearchQuery query){
    logger.info("Input : "+ query);
    if(Arrays.asList(originAirportShutdownList.split(","))
        .contains(query.getOrigin())){
        logger.info("The origin airport is in shutdown state");
        return new ArrayList<Flight>();
    }
    return searchComponent.search(query);
}
```

按照上面的代码对搜索方法进行修改后, 就可以读取 originAirportShutdownList, 看看请求的起点是否在关闭列表中。如果符合, 搜索结果就会直接返回空值而不是执行搜索算法。

(13) 启动 Config 服务器, 然后启动搜索微服务。一定要确保 RabbitMQ 正常运行。

(14) 修改 chapter5.website 项目, 匹配 bootstrap.properties 的下列内容, 从而使

用 Config 服务器：

```
spring.application.name=test-client
server.port=8001
spring.cloud.config.uri=http://localhost:8888
```

(15) 修改 Application.java 中 CommandLineRunner 的 run 方法，以 SEA 作为航班起点进行查询：

```
SearchQuery = new SearchQuery("SEA","SFO","22-JAN-16");
```

(16) 运行 chapter5.website 项目。CommandLineRunner 会返回一个空的航班列表。下面的消息会显示在服务器上：

```
The origin airport is in shutdown state
```

处理配置的改变

这一节会详细介绍，如何将属性的变化传播出去。

(1) 将 search-service.properties 文件的属性改成：

```
originairports.shutdown:NYC
```

提交修改到 Git 仓库。刷新 Config 服务器的 URL（http://localhost:8888/search-service/default），看看属性的变化有没有显示出来。如果一切顺利，我们在页面上能看到属性的变化。后续的请求会强制 Config 服务器重新从仓库中读取属性文件。

(2) 重启网站项目，观察 CommandLineRunner 的执行。注意，不需要重启搜索微服务和 Config 服务器。服务器还是会像之前一样返回空的航班信息，依旧显示如下消息：

```
The origin airport is in shutdown state
```

这意味着修改并没有通知到搜索服务，服务仍然是在旧版本的配置属性下运行。

(3) 为了强制重新载入配置属性，调用搜索微服务的/refresh 端，实际上这是 Actuator 的刷新端，下面的命令会向/refresh 端发送一个空的 POST 请求：

```
curl -d {} localhost:8090/refresh
```

(4) 重新启动网站项目，观察 CommandLineRunner 的执行。这次我们就会看到以 SEA 为起点的航班列表。注意：如果 Booking 预订服务没有启动，项目可能会报错。/refresh 端会刷新本地的配置属性缓存，从 Config 服务器重新载入刷新的值。

传播配置修改的 Spring Cloud Bus

通过前面介绍的方法，即便不重启微服务，也可以修改配置参数。当整个项目只有 1~2 个实例运行的时候，这是很好的。但是如果有很多实例呢？举个例子，如果有 5 个微服务，我们就不得不对每个微服务/refresh，这无疑是个负担，如图 5-10 所示。

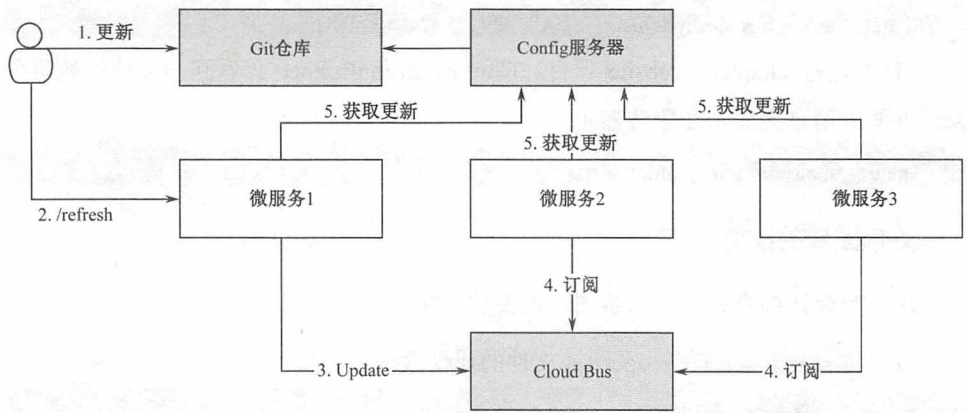


图 5-10

Spring Cloud Bus 可以很好地解决这个问题。它提供了一种不需要知道实例的数量和位置，就可以在多个实例间刷新配置的机制。当一个微服务有多个实例在运行，或者有多个不同种类的微服务在运行时，Spring Cloud Bus 显得尤为方便。它是通过单个消息的 Broker 来连接所有服务实例来实现这样神奇的功能的。每一个实例都订阅了修改时间，在需要的时候会刷新本地的配置。调用任何一个实例的/bus/refresh 端，都会触发这种刷新机制。然后修改信息会通过 Cloud Bus 和公共消息中间方传播出去。

在这个例子中，RabbitMQ 就是 AMQP 消息中间方，按下列步骤进行操作：

(1) 在 chapter5.search 项目的 pom.xml 文件中，添加对于 Cloud Bus 的依赖：

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

(2) 搜索微服务也需要连接到 RabbitMQ，这个已经在 `search-service.properties` 中提供好了。

(3) 重新构建和启动搜索微服务。我们在命令行中运行两个搜索微服务的实例：

```
java -jar -Dserver.port=8090 search-1.0.jar
java -jar -Dserver.port=8091 search-1.0.jar
```

这两个搜索服务的实例分别在 8090 和 8091 端口运行。

(4) 重新运行网站项目。这一步只是保证一切都顺利工作。搜索服务这次应该返回一架航班。

(5) 现在，用下面的值更新 `search-service.properties`，并提交到 Git：

```
originairports.shutdown:SEA
```

(6) 运行下面的命令。注意：我们的 Bus 和之前的一个实例运行在相同的端口 8090 上：

```
curl -d {} localhost:8090/bus/refresh
```

(7) 我们马上就在两个实例中收到下面的消息：

```
Received remote refresh request. Keys refreshed [originairports.
shutdown]
```

Bus 端从内部给消息 Broker 发送了一条消息，被所有实例消费，然后重新加载它们的属性文件。也可以用声明应用名的方式，将修改推送给特定的应用：

```
/bus/refresh?destination=search-service:**
```

我们也可以通过制定属性名，刷新特定的属性。

为 Config 服务器设置高可用

前面的几节探讨了如何建立 Config 服务器，并且实现配置属性的实时刷新。然而，在这样的架构中，Config 服务器是单点故障的。

前面几节建立的默认架构中，有 3 个单点故障。其中一个 Config 服务器本身的可用性，第 2 个是 Git 仓库，第 3 个是 RabbitMQ 服务器。

图 5-11 展示了 Config 服务器的高可用架构。

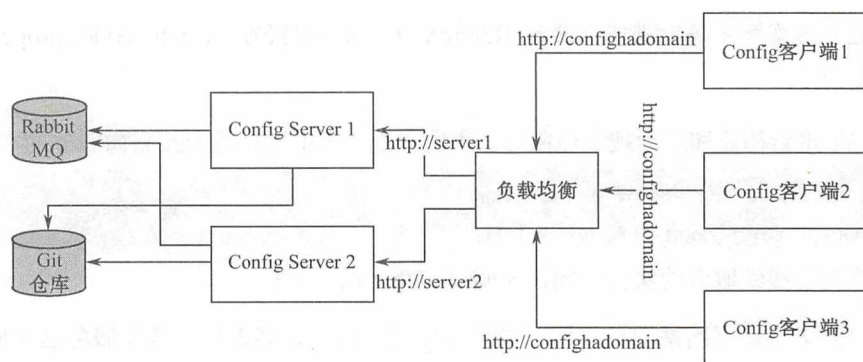


图 5-11

整个架构的机理是这样的：

Config 服务器需要高可用，因为一旦 Config 服务器不可用，服务就无法启动。因此，为了高可用，我们就需要备用的 Config 服务器。因此一旦服务启动之后，就算 Config 服务器不可用了，应用也是可以继续运行的。在这种情况下，服务会在最后已知的配置状态下继续运行。所以 Config 服务器的可用性并不像微服务的可用性要求那么严苛。

为了让 Config 服务器高可用，我们需要给 Config 服务器配置多个实例。由于 Config 服务器是无状态的 HTTP 服务，Config 服务器的多个实例是可以并行运行的。Config 服务器采用了负载均衡的思路，微服务需要修改配置去引入多个 Config 服务器，但是 `Bootstrap.properties` 只能处理单个 Config 服务器地址。因此多个 Config 服务器必须运行在负载均衡器或者具有故障转移和回退功能的本地 DNS 后面，这样基于 DNS 或者负载均衡器能够满足高可用和能够处理故障的前提下，负载均衡器或者 DNS 服务器的 URL 会被配置在微服务的 `bootstrap.properties` 文件中，而不是 Config 服务器的地址。

在生产环境下，使用基于本地文件的 Git 仓库并不是一个好主意。配置服务器应该用高可用的内部或者外部 Git 服务备份。SVN 也可以考虑。

我们前面提到过，Config 服务器能用本地备份的一个配置文件跑起来，只有在 Config 服务器需要扩容的情况下，我们才需要一个高可用的 Git，因为需要从 Git 上面下载配置文件到扩容的实例中。但是它不像微服务或者 Config 服务器的可用性那

样要求那么严苛。

提示

创建高可用 GitLab 的例子可以查看：<https://about.gitlab.com/high-availability/>。

RabbitMQ 也要配置为高可用。RabbitMQ 的高可用只是用来动态推送配置的修改到所有实例。因为这更是一个脱机版的活动，而并不需要像组件一样高可用。

使用云服务或者本地配置的高可用 RabbitMQ 服务，都可以实现 RabbitMQ 的高可用。

提示

创建高可用 RabbitMQ 的文档可以参照：<https://www.rabbitmq.com/ha.html>。

监控 Config 服务器的健康

Config 服务器本质上是一个配置了 actuator 的 Spring Boot 应用。因此，对于 Config 服务器而言，所有的 actuator 端都是可用的。服务器的健康状况可以用下面的 URL 进行监控：<http://localhost:8888/health>。

配置文件的 Config 服务器

实际中我们可能需要一个完整的外部配置文件（如 logback.xml）。Config 服务器为我们提供了配置和存储这种文件的机制。只要使用下面的 URL 就可以实现：

```
{name}/{profile}/{label}/{path}
```

Name、profile、label 的含义和前面一样。Path 指的是如 logback.xml 的文件名称。

完成最后的修改

构建完这个功能之后，我们就可以完成 BrownField 航空的 PSS 系统。我们现在需要做的是在所有服务中使用配置服务器。第 5 章中的所有微服务都要做相似的修改，到 Config 服务器中去请求配置参数。

下面几点是修改的关键点：

- 预订组件的 Fare 服务的 URL 也会被暴露：

```
private static final String FareURL = "/fares";
@Value("${fares-service.url}")
private String fareServiceUrl;
Fare = restTemplate.getForObject(fareServiceUrl+FareURL +"/
get?flightNumber="+record.getFlightNumber()+"&flightDate="+record.
getFlightDate(),Fare.class);
```

正如前面的代码片段所示，Fare 服务的 URL 通过新的属性得到：fares-service.url。

- 我们暂时不会暴露搜索、预订、登记服务的队列名。在本章的后面部分，会使用 Spring Cloud Streams 来更改它们。

一个声明式的 REST 客户端 Feign

在预订微服务中，通过 RestTemplate 对 Fare 微服务发生了同步调用。当我们使用 RestTemplate 的时候，URL 参数由程序自动构建，数据被发送到其他服务。在更复杂的场景下，我们必须深入 RestTemplate 提供的 HTTP API 的细节，甚至是一个低等级的 API。

Feign 是一个 Spring Cloud Netflix 库，它提供相比基于 REST 的服务调用更高度抽象化的概念。Spring Cloud Feign 的工作方式是声明式的。当使用 Feign 的时候，我们要在客户端写声明式的 REST 服务接口，并且使用这些接口进行编程。开发者不需要担心接口的具体实现。这会在运行时由 Spring 动态生成。通过这种声明式的方法，开发者不需要深入 RestTemplate 提供的 HTTP 级别的接口细节。

下面是预订微服务中，调用 Fare 服务的代码片段：

```
Fare fare = restTemplate.getForObject(FareURL +"/
get?flightNumber="+record.getFlightNumber()+"&flightDate="+record.getFlightDate(),Fare.cl
ass);
```

为了使用 Feign，首先我们需要在 pom.xml 中添加对 Feign 的依赖：

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-feign</artifactId>
```

```
</dependency>
```

对于一个新的 Spring Stater 项目，Feign 可以在启动库选择窗口选择，或者从 <http://start.spring.io/> 选择。在 Cloud Routing 下也可以选择，如图 5-12 所示。

Cloud Routing

- ☐ Zuul
Intelligent and programmable routing with spring-cloud-netflix Zuul
- ☐ Ribbon
Client side load balancing with spring-cloud-netflix and Ribbon
- ☐ Feign
Declarative REST clients with spring-cloud-netflix Feign

图 5-12

下一步是创建一个新的 FareServiceProxy 接口。这个接口会起到 Fare 服务代理接口的作用：

```
@FeignClient(name="fares-proxy", url="localhost:8080/fares")
public interface FareServiceProxy {
    @RequestMapping(value = "/get", method=RequestMethod.GET)
    Fare getFare(@RequestParam(value="flightNumber") String
        flightNumber, @RequestParam(value="flightDate") String
        flightDate);
}
```

FareServiceProxy 接口有一个 @FeignClient 注解。这个注解告诉 Spring 创建一个基于已提供接口的 REST 客户端。这个值可以是服务 ID，也可以是逻辑名称。url 指明了目标服务运行时的 URL。名称和值都不是必须的。上例中因为有 url 属性，name 属性便无关紧要了。

使用这个服务代理调用 Fare 服务。在预订微服务中，我们必须告诉 Spring，在 Spring Boot 应用中存在 Feign 客户端，并且要去扫描。在 BookingComponent 的类级中添加 @EnableFeignClients 注解就可以实现这一功能。另外，我们也可以提供需要扫描的包名。

修改 BookingComponent 的调用部分。这个跟调用另一个 java 接口一样简单：

```
Fare = fareServiceProxy.getFare(record.getFlightNumber(), record.
    getFlightDate());
```


重新运行预订微服务，看看发生的变化。

FareServiceProxy 接口中的 Fare 服务的 URL 是硬编码的：

```
url="localhost:8080/fares".
```

我们暂时会让它维持原状，本章的后面部分会做出修改。

用于负载均衡的 Ribbon

在前面的设置中，我们一直在运行微服务的单个实例。在客户端和服务之间的调用，URL 都是硬编码的。在现实世界中，这种做法是不可取的，服务实例应该不止一个。理想状态下，如果有多个实例，我们应该使用负载均衡器或者本地的 DNS 服务器来抽象实际实例的位置，然后在客户端中配置一个别名或者负载均衡器地址。负载均衡器会接收别名，并且在一个可用的实例中进行解析。通过这种方法，我们可以在负载均衡器后面配置很多实例。它也可以帮助我们处理对客户端透明的服务器故障。

上面所说的功能是通过 Spring Cloud Ribbon 来实现的。Ribbon 是客户端的负载均衡器，可以在一系列服务器间实现循环负载均衡。在 Ribbon 库中，也可能有很多其他的负载均衡算法。Spring Cloud 提供了声明式的配置和使用 Ribbon 客户端的方法。

如图 5-13 所示，Ribbon 客户端向 Config 服务器获取可用的微服务实例列表，默认情况下进行循环负载均衡算法。

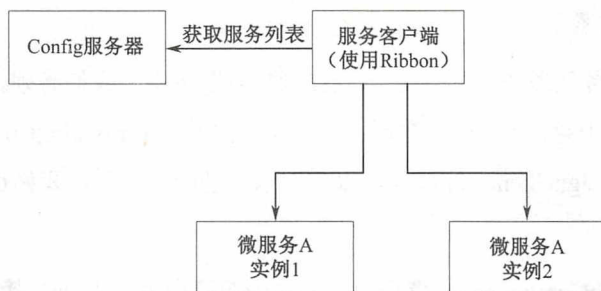


图 5-13

为了使用 Ribbon 客户端，我们需要在 pom.xml 中添加如下依赖：

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-ribbon</artifactId>
</dependency>
```

如果项目是从头开发的，可以在 Spring Starter 库中选择 Ribbon，或者从 <http://start.spring.io/> 选择。Cloud Routing 中也可以选择 Ribbon，如图 5-14 所示。

Cloud Routing

- ☐ Zuul
Intelligent and programmable routing with spring-cloud-netflix Zuul
- ☐ Ribbon
Client side load balancing with spring-cloud-netflix and Ribbon
- ☐ Feign
Declarative REST clients with spring-cloud-netflix Feign

图 5-14

更新预订微服务的配置文件，booking-service.properties，包含一个新的属性来保存付款微服务列表：

```
fares-proxy.ribbon.listOfServers=localhost:8080,localhost:8081
```

现在我们回过头去修改之前写的 FareServiceProxy 类，让它使用 Ribbon 客户端。我们注意到，@RequestMapping 注解的值从 /get 修改为 /fare/get，这样我们就可以把主机名和端口轻易地移动到配置中。

```
@FeignClient(name="fares-proxy")
@RibbonClient(name="fares")
public interface FareServiceProxy {
    @RequestMapping(value = "/fares/get", method=RequestMethod.GET)
```

现在，我们可以运行 Fare 微服务的两个实例，一个在 8080 端口，一个在 8081 端口：

```
java -jar -Dserver.port=8080 fares-1.0.jar
java -jar -Dserver.port=8081 fares-1.0.jar
```

运行预订微服务。启动之后，CommandLineRunner 自动在第一个服务器中插入一条预订记录。

运行整个网站项目的时候，会调用预订服务。这个请求会发送到第二个服务器上。

在预订服务中，我们会看到下面的路径，这意味着有两台服务器可用：

```
DynamicServerListLoadBalancer: {NFLoadBalancer:name=fares-proxy,current
list of Servers=[localhost:8080, localhost:8081],Load balancer stats=Zone
stats: {unknown=[Zone:unknown; Instance count:2; Active connections
count: 0; Circuit breaker tripped count: 0; Active connections per
server: 0.0;]}
```

注册和发现的 Eureka

目前为止，我们实现了从 Config 服务器配置参数和多个实例之间的负载均衡。

基于 Ribbon 的负载均衡可以满足大部分的微服务要求。然而，在某些场景下还是不足的：

- 如果有大量微服务，想要优化微服务的利用率我们必须动态改变服务实例的数量和相关的服务器。但是在配置文件中提前预测和配置服务器的 URL 是很难的。
- 高伸缩性的微服务在进行云部署的时候，考虑到云环境的弹性，静态的注册和发现并不适合。
- 在云部署的场景下，IP 地址无法预知，很难在文件中静态配置。每次地址发生改变时，我们必须更新配置文件。

Ribbon 在一定程度上解决了这个问题。我们可以动态修改服务实例，但是一旦需要修改或者关闭服务实例，我们必须手动更新 Config 服务器。尽管配置的修改会自动传播到所有必要的实例，手动的配置修改面对大规模部署是杯水车薪。大规模的部署，只有自动化才是王道。

为了弥补这个不足，微服务应该通过动态注册其服务可用性，自己管理生命周期，并且应该被消费者自动发现。

理解动态服务注册和发现

动态注册是从服务提供者的角度而言的。当一个新的服务启动的时候，它自动在服务注册中心显示。相似地，当一个服务停止，它也会自动把自己从服务注册中心去掉。注册中心总是与最新的可用服务信息和元数据保持一致。

动态发现是从服务消费者的角度而言的。动态发现的概念是，当客户端向服务注册中心请求当前的服务拓扑结构时，会请求相应的服务。通过这种方法，URL 是从服务注册中心动态获取的，而不是静态写死到配置文件中。

客户端会保存一份注册中心的数据到本地缓存，这样方便更快速地连接。一些注册中心也允许客户端监控它们感兴趣的节点。通过这种方法，服务注册中心一旦发生状态改变，就会通知到相关方从而避免使用旧的数据。

实现动态服务注册和发现有多种方式。Spring Cloud 为我们提供了 Netflix Eureka、ZooKeeper 和 Consul，如图 5-15 所示，Etcd 是 Spring Cloud 之外的另一个可以实现动态服务注册和发现的服务注册中心。本章中，我们会详细探讨 Eureka 的实现。

理解 Eureka

Spring Cloud Eureka 也来自 Netflix OSS。Spring Cloud 项目提供了对 Spring 友好的声明式方法，可以将 Eureka 和基于 Spring 的应用完美整合。Eureka 主要用于自我注册、动态发现及负载均衡。Eureka 内部使用 Ribbon 实现负载均衡。

Cloud Discovery

- ☐ Eureka Discovery
Service discovery using spring-cloud-netflix and Eureka
- ☐ Eureka Server
spring-cloud-netflix Eureka Server
- ☐ Zookeeper Discovery
Service discovery with Zookeeper and spring-cloud-zookeeper-discovery
- ☐ Cloud Foundry Discovery
Service discovery with Cloud Foundry
- ☐ Consul Discovery
Service discovery with Hashicorp Consul

图 5-15

如图 5-16 所示, Eureka 包括一个服务器组件和一个客户端组件。服务器端组件是注册中心, 所有的微服务在这里注册, 注册信息通常是服务的身份和 URL。微服务使用 Eureka 客户端来发布服务, 消费组件也会使用 Eureka 客户端来发现服务实例。

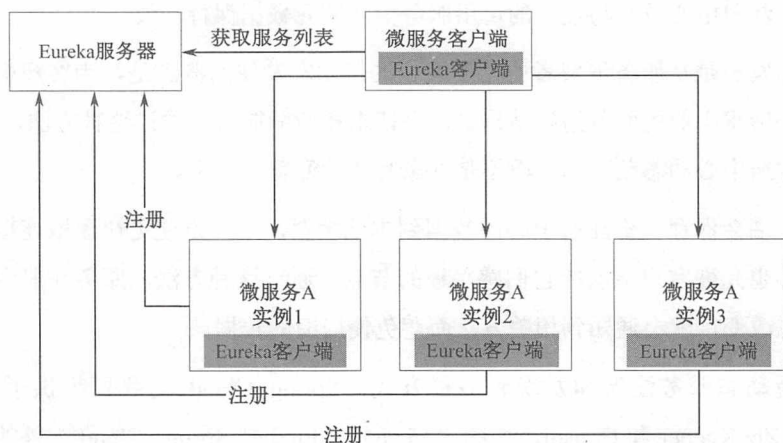


图 5-16

微服务启动的时候, 会告知 Eureka 服务器其存在及相关信息, 一旦注册成功, 服务每 30s 向注册中心发送 ping 请求去更新其注册信息。如果一个服务不能更新信息, 注册中心就会剔除该服务。注册中心的信息会复制到所有的 Eureka 客户端, 客户端的每次请求都必须通过 Eureka 服务器。Eureka 客户端从服务器取到注册信息并在本地做缓存。这之后, 客户端用这些信息找到其他服务。这些信息后面都是进行周期性 (每 30s) 的增量更新。

当一个客户端想要连接到微服务, Eureka 客户端将基于被请求的服务 ID, 提供该服务的可用实例的列表。Eureka 服务器端是对域 (zone) 敏感的。在注册服务的时候, 域信息也是可以提供的。当一个客户端请求一个服务实例时, Eureka 服务会尽量查找同一域 (zone) 下运行中的服务。然后 Ribbon 客户端在 Eureka 客户端提供的可用服务实例之间做负载均衡。Eureka 客户端和服务器之间通过 REST 和 JSON 进行信息传递。

设置 Eureka 服务器

在这一节中, 我们将会介绍设置 Eureka 服务器的方法。

所有源代码可以在 chapter5.eureka.server 文件中看到。注意，Eureka 服务器的注册和刷新周期为 30s。因此，运行服务和客户端的时候，需要等待 40~50s。

(1) 建立新的 Spring Starter 项目，选择 Config Client、Eureka Server、Actuator，如图 5-17 所示。

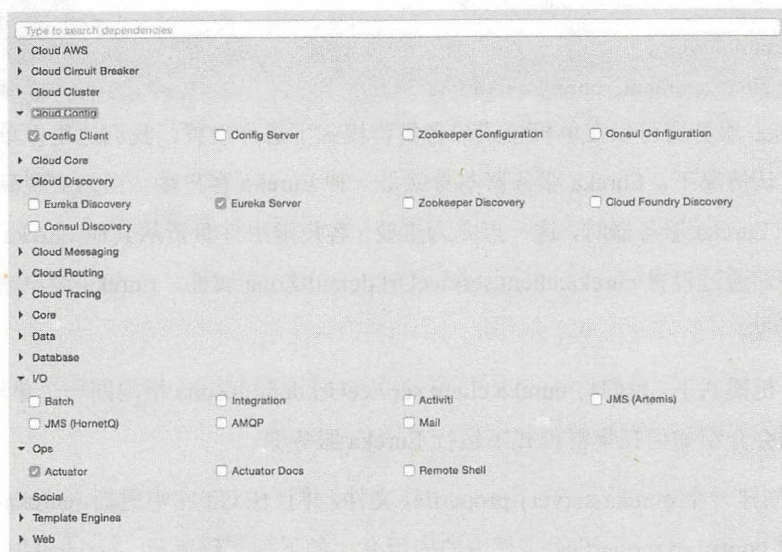


图 5-17

Eureka 服务器的项目结构如图 5-18 所示。

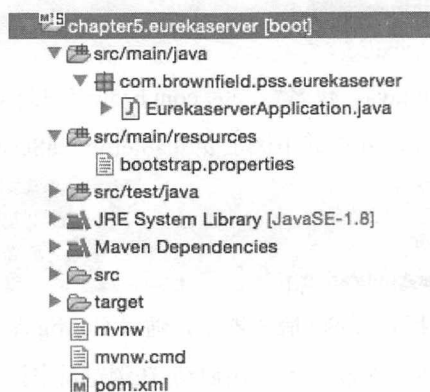


图 5-18

需要注意的是，主程序名称为 `EurekaServerApplication.java`。

(2) 因为使用 Config 服务器，需要将 `application.properties` 重命名为 `bootstrap.properties`。和我们之前的做法一样，在 `bootstrap.properties` 文件中配置 Config 服务器的详细信息，这样就可以定位 Config 服务器实例。`bootstrap.properties` 文件内容如下：

```
spring.application.name=eureka-server1
server.port:8761
spring.cloud.config.uri=http://localhost:8888
```

Eureka 服务器可以在单机模式或者集群模式下进行设置。我们这里以单机模式为例。默认情况下，Eureka 服务器本身就是一种 Eureka 客户端。当为实现高可用而采用多个 Eureka 服务器时，这一点尤为重要。客户端组件负责从其他 Eureka 服务器同步状态。通过设置 `eureka.client.serviceUrl.defaultZone` 属性，Eureka 客户端可以被其同类接收。

在单机模式下，我们将 `eureka.client.serviceUrl.defaultZone` 指向同一个单机实例。后面我们会介绍如何在集群模式下运行 Eureka 服务器。

(3) 创建一个 `eureka-server1.properties` 文件，并且在 Git 库中更新。`eureka-server1` 是应用的 `bootstrap.properties` 文件中的应用名。如下列代码所示，`serviceUrl` 指向同一个服务器。一旦添加了下面的属性，将文件加入 Git 同步：

```
spring.application.name=eureka-server1
eureka.client.serviceUrl.defaultZone:http://localhost:8761/eureka/
eureka.client.registerWithEureka:false
eureka.client.fetchRegistry:false
```

(4) 修改 `Application.java`，包名修改为 `com.brownfield.pss.eurekaserver`，类命名为 `EurekaServerApplication`，并在其中添加 `@EnableEurekaServer` 注解。

```
@EnableEurekaServer
@SpringBootApplication
public class EurekaServerApplication {
```

(5) 现在我们可以启动 Eureka 服务器了，确保 Config 服务器运行中。在应用上右键选择 `Run As | SpringBoot App`，一旦应用启动，在浏览器中打开 `http://localhost:8761`，看看 Eureka 控制台。

(6) 在控制台，注意到当前并没有以 Eureka 注册的实例。因为没有服务随着

Eureka client 的激活而被发现，在这个节点列表是空的。

(7) 微服务的任何修改都会使用 Eureka 服务进行动态注册和发现。为了实现这一点，首先我们要在 pom.xml 中加入对于 Eureka 的依赖。如果这些服务是使用 Spring Starter 来创建的，请选择 Config Client、Actuator、Web 和 Eureka 发现客户端，如图 5-19 所示。

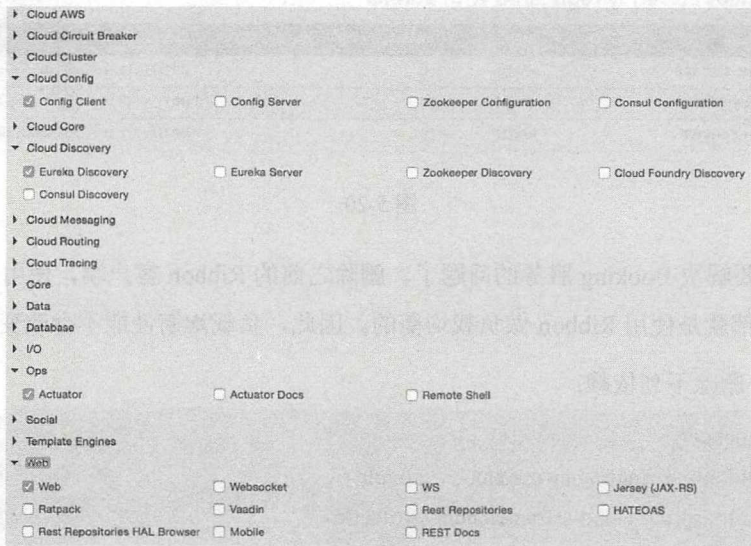


图 5-19

(8) 因为我们要修改微服务，在所有微服务的 pom.xml 中添加下列依赖：

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

(9) 下列的属性已经被添加到所有微服务（在各自微服务的 config-repo 的配置文件 中）。这将会有助于微服务连接 Eureka 服务器，一旦完成后，提交到 Git。

```
eureka.client.serviceUrl.defaultZone: http://localhost:8761/
eureka/
```

(10) 在微服务各自的 Spring Boot 主 class 中添加@EnableDiscoveryClient 注解。这会要求 Spring Boot 在服务启动的时候注册它们的可用性。

(11) 启动除了 Booking 之外的所有服务。因为我们在 Booking 服务中使用了 Ribbon 客户端，当我们在 class 路径中加入 Eureka 客户端，服务的整体表现将会有所区别。稍后我们会修复这个问题。

(12) 跳转到 Eureka URL (<http://localhost:8761>)，您可以看到正在运行的 3 个实例，如图 5-20 所示。

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CHECKIN-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:checkin-service:8070
FARES-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:fares-service:8080
SEARCH-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:search-service:8090

图 5-20

现在要解决 Booking 服务的问题了。删除之前的 Ribbon 客户端，使用 Eureka。Eureka 内部就是使用 Ribbon 做负载均衡的。因此，负载均衡性能不会改变。

(13) 删除下列依赖：

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-ribbon</artifactId>
</dependency>
```

(14) 在 FareServiceProxy 类中，删除@RibbonClient(name="fares")注解。

(15) 更新@FeignClient(name="fares-service")注解使其匹配真实的 Fare 微服务 ID。这种情况下，fares-service 是 Fare 微服务中 bootstrap.properties 文件内配置的服务 ID，也是 Eureka 发现客户端发送到服务器的名称。服务 ID 是服务注册在 Eureka 服务器的标识。

(16) 从 booking-service.properties 中删除服务器列表。有了 Eureka，我们可以从 Eureka 服务器动态发现这个列表：

```
fares-proxy.ribbon.listOfServers=localhost:8080, localhost:8081
```

(17) 启动 Booking 服务。您会看到 CommandLineRunner 已经成功创建了预订，这个过程中包括使用 Eureka 的发现机制调用 Fare 服务，跳转到 URL 可以看到所有注册的服务，如图 5-21 所示。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
BOOK-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:book-service:8060
CHECKIN-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:checkin-service:8070
FARES-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:fares-service:8080
SEARCH-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:search-service:8090

图 5-21

(18) 修改网站项目的 `bootstrap.properties` 文件，使用 Eureka 而不是直接连接到服务实例。这种情况下我们不会使用 Feign 客户端。取而代之的是，为了强调作用，我们会使用负载均衡的 RestTemplate。

```
spring.application.name=test-client
eureka.client.serviceUrl.defaultZone: http://localhost:8761/
eureka/
```

(19) 在 Application 类中加入 `@EnableDiscoveryClient` 注解，这样客户端可以被 Eureka 发现。

(20) 修改 Application.java 和 BrownFieldSiteController.java。添加 3 个 RestTemplate 实例。我们可以使用 `@Loadbalanced` 注解来开启 Eureka 和 Ribbon。由于 RestTemplate 无法自动注入，因此，我们必须提供如下的配置入口：

```
@Configuration
class AppConfiguration {
    @LoadBalanced
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

@Autowired
RestTemplate searchClient;

@Autowired
RestTemplate bookingClient;

@Autowired
RestTemplate checkInClient;
```

(21) 我们使用 `RestTemplate` 实例来调用微服务。用注册在 Eureka 服务器的服务 ID 替换硬编码的 URL。在下面的代码中, 我们使用 `search-service`、`book-service` 和 `checkin-service` 等服务名, 而不是显式的主机名称和端口。

```
Flight[] flights = searchClient.postForObject("http:// search-service/search/get", searchQuery,
Flight[].class);
long bookingId = bookingClient.postForObject("http://book-service/booking/create", booking,
long.class);
long checkinId = checkInClient.postForObject("http://checkin-service/checkin/create", checkIn,
long.class);
```

(22) 现在我们运行客户端和整个 Web 项目。如果一切顺利, Web 项目的 `CommandLineRunner` 会成功实现搜索、预订和签到功能。同样, 我们可以在浏览器中通过 `http://localhost:8001` 进行测试。

Eureka 的高可用

在前面的例子中, 单机模式下只有一个 Eureka 服务器。对于真正的生产系统这是远远不够的。

Eureka 客户端连接到服务器, 取到注册中心的信息, 并在本地储存在缓存中, 客户端一直是用本地缓存来工作的。Eureka 客户端周期性地检查服务器有没有状态改变。一旦发生改变, 客户端会从服务器下载相应的修改, 并在缓存中做更新。如果无法连接到服务器, Eureka 客户端仍然可以基于客户端缓存中的数据, 与最后已知的服务器状态进行工作, 但这可能会很快导致状态过期问题。

这一节将会介绍 Eureka 服务器的高可用, 整个架构如图 5-22 所示。

Eureka 服务器是建立在对等数据同步机制上的。运行的状态信息并不存储在数据库, 而是在内存中进行缓存。高可用的实现要考虑 CAP 理论中的可用性、分区容错性, 放弃一致性。因为 Eureka 服务器的实例之间是使用异步机制进行同步的, 服务器实例的状态无法保证永远一致。对等同步是通过互相之间指向 `serviceUrl` 来实现的。如果有不止一个 Eureka 服务器, 每一个都必须连接到至少一个对等服务器。由于状态是在所有对等服务器间复制的, Eureka 客户端可以连接到任意一台可用的 Eureka 服务器。

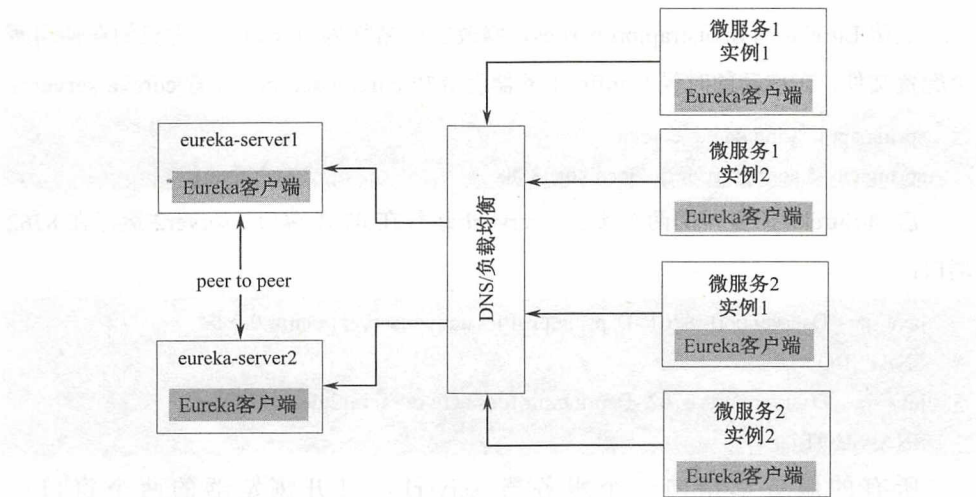


图 5-22

实现 Eureka 高可用的最佳方式是使用多个 Eureka 服务器集群，运行在负载均衡器或者本地 DNS 后面。客户端总是使用 DNS/负载均衡器连接服务器的。在运行的时候，负载均衡器负责选择正确的服务器。负载均衡器的地址会被提供给 Eureka 客户端。

这一节，我们会展示如何在高可用集群中运行两个 Eureka 服务器。首先我们要定义两个属性文件，eureka-server1 和 eureka-server2，这两个是对等服务器，如果其中一个失效，另一个就会顶替上来。每一个对等服务器也会作为另一个的客户端，这样就可以同步状态。两个属性文件如下面的代码片段所示。上传和提交这些属性到 Git 仓库。

客户端 URL 指向彼此，形成一个如下面配置所示的对等网络：

```
eureka-server1.properties
eureka.client.serviceUrl.defaultZone:http://localhost:8762/eureka/
eureka.client.registerWithEureka:false
eureka.client.fetchRegistry:false
eureka-server2.properties
eureka.client.serviceUrl.defaultZone:http://localhost:8761/eureka/
eureka.client.registerWithEureka:false
eureka.client.fetchRegistry:false
```

上传 Eureka 的 bootstrap.properties, 修改应用名称为 eureka。因为我们在使用两个配置文件, 在启动的时候 Config 服务器会寻找 eureka-server1 或者 eureka-server2:

```
spring.application.name=eureka
spring.cloud.config.uri=http://localhost:8888
```

启动 Eureka 服务器的两个实例, server1 运行在 8761 端口, server2 运行在 8762 端口:

```
java -jar -Dserver.port=8761 -Dspring.profiles.active=server1 demo-0.0.1-SNAPSHOT.jar
java -jar -Dserver.port=8762 -Dspring.profiles.active=server2 demo-0.0.1-SNAPSHOT.jar
```

所有的服务都指向一个服务器 server1, 打开浏览器的两个窗口: <http://localhost:8761> 和 <http://localhost:8762>。

启动所有的微服务。8761 端口的服务器立刻就能感知到改变, 另一个则延迟了 30s。因为两个服务器在一个集群中, 它们的状态是同步的。如果我们让这些服务器处在负载均衡器或者 DNS 后面, 客户端永远都能与一个可用的服务器连接。

完成上述步骤之后, 回到单机模式完成接下来的任务。

API 网关——Zuul 代理

在大多数微服务实现中, 内部微服务不会暴露在外面。它们是隐秘的服务。一系列公共服务将会使用 API 网关暴露给客户端。这样做的原因如下:

- 客户端只需要一系列特定的微服务。
- 如果有客户端特定的策略, 在一个地方应用比多个地方容易。一个现实的例子就是跨源访问策略。
- 在服务端很难实现客户端特定的转换。
- 如果有必要数据聚合, 尤其是在带宽有限的环境中避免多个客户端调用, 中间就需要一个网关。

Zuul 是一个简单的网关服务，或者可以理解为一个边缘服务。它可以很好地适应上述情景。Zuul 也来自微服务产品的 Netflix 家族。与许多企业级 API 网关产品不同的是，Zuul 为开发者提供完整的配置或者基于特定需求的程序的控制，如图 5-23 所示。

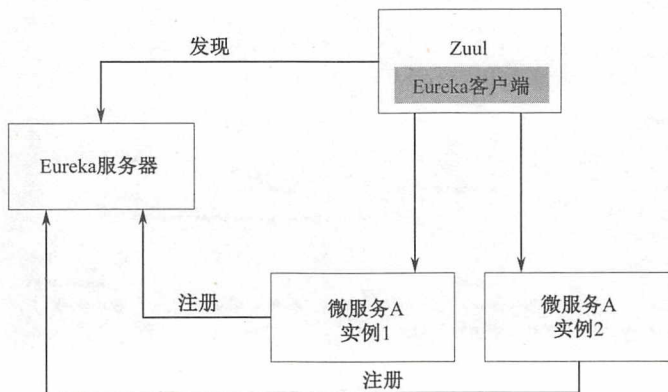


图 5-23

Zuul 代理内部使用 Eureka 服务器实现服务的发现，Ribbon 则负责实现微服务实例之间的负载均衡。

Zuul 代理能够实现路由、监控、管理弹性、安全等功能。简单地说，我们可以把 Zuul 当作一个反向代理服务。我们甚至可以在 API 层覆盖基本服务来改变它们的表现。

建立 Zuul

不像 Eureka 服务器和 Config 服务器，在典型的部署过程中，Zuul 针对特定的微服务。然而，有些部署中，一个 API 网关覆盖很多微服务。这种时候，我们应该添加 Zuul 给每个微服务：Search、Booking、Fare 和 Check-in。

所有源代码可以在 chapter5.*-apigateway 项目文件夹下找到。

(1) 一个个转换微服务。首先是 Search 的 API 网关。新建一个 Spring Starter 项目，选择 Zuul、Config Client、Actuator 和 Eureka Discovery，如图 5-24 所示。

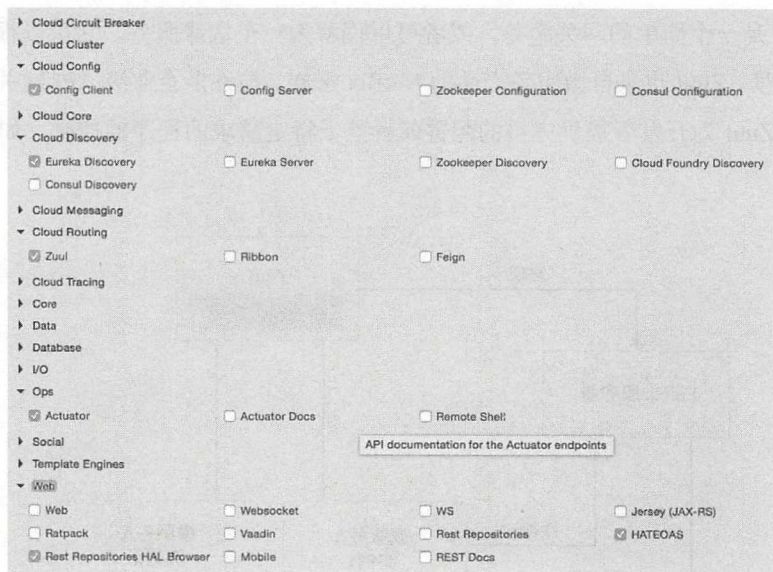


图 5-24

Search-apigateway 项目结构如图 5-25 所示。

(2) 下一步是整合 API 网关、Eureka 和 Config 服务器。新建 search-apigateway.property 文件，加入下面的内容，提交到 Git 仓库。

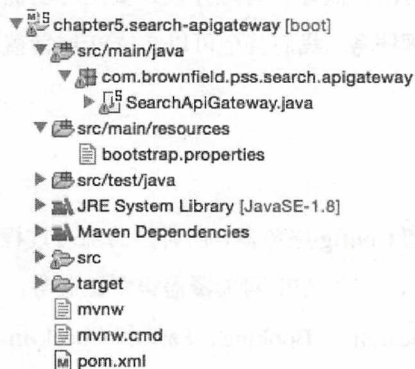


图 5-25

这个配置也设置了如何转发流量。这种情况下，任何发送到 API 网关的/api 的请求都应该被发送到 search-service:

```
spring.application.name=search-apigateway
```

```
zuul.routes.search-apigateway.serviceId=search-service
zuul.routes.search-apigateway.path=/api/**
eureka.client.serviceUrl.defaultZone:http://localhost:8761/eureka/
```

search-service 是 Search 服务的服务 ID，会在 Eureka 服务器中被解析。

(3) 更新 search-apigateway 的 bootstrap.properties 文件。配置文件和之前几节的结构一样——服务名、端口、Config 服务器 URL。

```
spring.application.name=search-apigateway
server.port=8095
spring.cloud.config.uri=http://localhost:8888
```

(4) 修改 Application.java。在这种情况下，包名和类名分别应该被修改为 com.brownfield.pss.search.apigateway 和 SearchApiGateway。添加@EnableZuulProxy 告诉 Spring Boot 这是 Zuul 代理：

```
@EnableZuulProxy
@EnableDiscoveryClient
@SpringBootApplication
public class SearchApiGateway {
```

(5) 运行 Spring Boot App。在此之前，确认 Config 服务器、Eureka 服务器和 Search 微服务都在运行。

(6) 修改 Web 项目的 CommandLineRunner 和 BrownFieldSiteController，来使用 API 网关。

```
Flight[] flights = searchClient.postForObject("http://search-apigateway/api/search/get", search
Query, Flight[].class);
```

在这种情况下，Zuul 代理起到了反向代理的作用为消费者访问的所有微服务提供代理，在前面的例子中，Zuul 代理并没有添加任何值，我们只是把发送过来的请求传给相应的后端服务。

当我们有如下的一个或多个需求时，Zuul 尤其有用：

- 在网关而不用在每一个微服务实施强制身份验证和其他安全策略。网关可以在把请求发送给相关的后台服务之前，处理安全策略、token 等。它也可以基于某些业务策略做基础的拒绝，如阻止从黑名单用户发来的请求。
- 可以在网关层执行业务了解和监控。收集实时数据信息，然后推送给外部

系统进行分析。在一个地方而不需要横跨许多微服务，这样做是很容易的。

- API 网关在需要基于细粒度控制的动态路由场景下尤为有用。我们来举 3 个例子帮大家理解这种场景。第一个是，基于业务的特定值（如“起点国家”）发送请求到不同的服务实例。第二个例子是，从一个地区发过来的请求都被发送到一个服务实例组。最后一个例子是，所有来自某一特定产品的请求被路由到一个服务实例组。
- API 网关的另一个应用场景是，处理负载分流和节流。例如，我们必须控制基于阈值的负载（如每天的请求数）。例如，控制来自低价值的第三方在线通道的请求。
- Zuul 网关对于细粒度负载均衡场景也很适用。Zuul、Eureka 客户端和 Ribbon 一起针对负载均衡需求提供了细粒度控制。因为 Zuul 的实现其实是一个 Spring Boot 应用，开发者对于负载均衡有完全控制权。
- Zuul 网关对于数据融合的需求场景也很有用，如果消费者想要更高级别的粗粒度服务，网关就应该在内部代表客户端通过调用不止一个服务来实现聚合数据。当客户端工作在低带宽环境下，这种做法尤其可行。

Zuul 也提供了大量的过滤器，分为前过滤器、路由过滤器、后过滤器及错误过滤器。顾名思义，它们在服务调用生命周期的不同阶段使用。Zuul 也为开发者提供了可以自定义过滤器的功能。为了实现自定义过滤器，继承 `ZuulFilter` 抽象类，实现下列方法：

```
public class CustomZuulFilter extends ZuulFilter{  
    public Object run(){}  
    public boolean shouldFilter(){}  
    public int filterOrder(){}  
    public String filterType(){}  
}
```

实现自定义过滤器之后，把这个类加入主 context。在本例中，添加到下面的 `SearchApiGateway` 类中：

```
@Bean  
public CustomZuulFilter customFilter() {  
    return new CustomZuulFilter();  
}
```


如我们之前所说，Zuul 代理是一个 Spring Boot 服务。我们可以编程实现自定义的网关。如下面代码所示，我们可以在网关中加入自定义微服务，反过来它可以调用后端服务。

```
@RestController
class SearchAPIGatewayController {
    @RequestMapping("/")
    String greet(HttpServletRequest req){
        return "<H1>Search Gateway Powered By Zuul</H1>";
    }
}
```

在前面的例子中，只是加入了一个新的端口，并且从网关返回一个值。后面我们会使用 `@Loadbalanced RestTemplate` 来调用后端服务。因为我们拥有完全控制权，我们也可以进行数据转换、数据融合等。我们也可以使用 Eureka API 得到服务器列表，实现完全独立的负载均衡或者流量整形机制，而不是采用 Ribbon 自带的负载均衡功能。

Zuul 的高可用

Zuul 是基于 Http 的无状态服务，因此，我们可以有尽可能多的 Zuul 实例，而且没有任何相关性或黏性需求。然而，Zuul 的高可用是很严格的，因为所有从消费者到提供者的流量都要经过 Zuul 代理。然而对于做重活的后端微服务，弹性扩（缩）容的要求却反而不那么严苛。

Zuul 的高可用架构由其使用场景决定。典型的使用场景有：

- 客户端 JavaScript MVC（如 Angular JS），从远端浏览器访问 Zuul 服务。
- 另一个微服务或者非微服务通过 Zuul 访问服务。

在某些情况下，客户端可能无法使用 Eureka 客户端库，如一个用 PL/SQL 写的遗产应用。组织策略不允许 Internet 客户端处理客户端的负载均衡。至于基于浏览器的客户端，有可用的第三方 Eureka JS 库。

总之，一切取决于客户端是否使用 Eureka 客户端库。基于此，我们有两种方式实现 Zuul 的高可用。

当客户端也是 Eureka 客户端时, Zuul 的高可用

在这种情况下, 因为客户端也是一个 Eureka 客户端, Zuul 可以像其他微服务一样配置。Zuul 用服务 ID 在 Eureka 中注册自己。然后客户端使用 Eureka 和服务 ID 解析 Zuul 实例。

如图 5-26 所示的 search-apigateway, Zuul 服务用服务 ID 在 Eureka 中注册自己。Eureka 客户端用 ID “search-apigateway” 请求服务列表。Eureka 服务器返回基于当前 Zuul 拓扑的服务器列表。Eureka 客户端从列表选择一个服务器并启动调用。

如我们之前所看到的, 客户端使用服务 ID 解析 Zuul 实例。下面的例子中, search-apigateway 就是注册在 Eureka 中的 Zuul 实例 ID:

```
Flight[] flights = searchClient.postForObject("http://search-apigateway/api/search/get", searchQuery, Flight[].class);
```

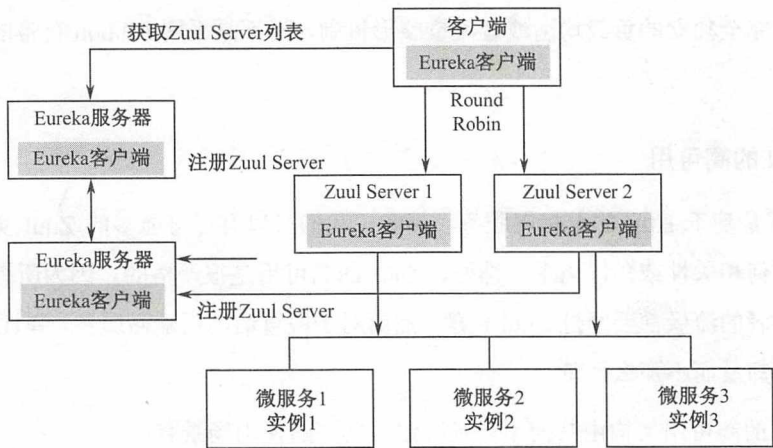


图 5-26

客户端不是 Eureka 客户端时的高可用

在这个例子中, 客户端不使用 Eureka 服务器来处理负载均衡。如图 5-27 所示, 客户端发送请求到负载均衡器, 负载均衡器轮流 (反过来) 定义正确的 Zuul 服务实例。在这种情况下, Zuul 实例运行在负载均衡器 (HAProxy 或者硬件负载均衡器 NetScaler) 后面。

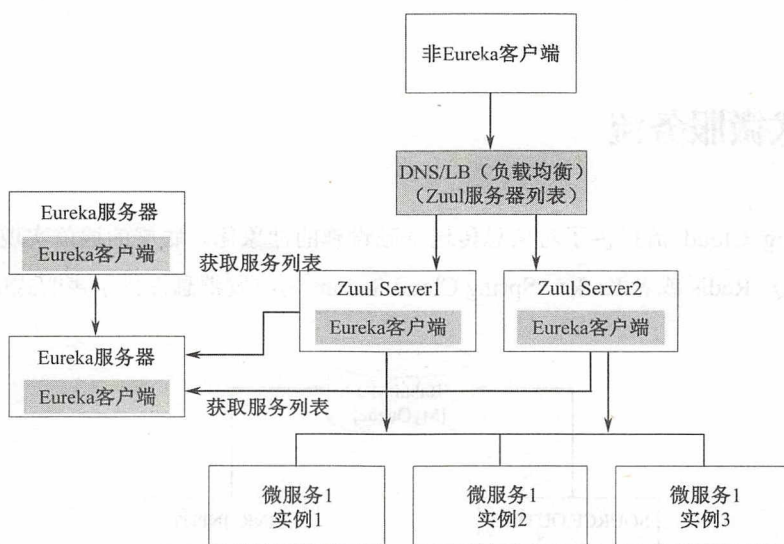


图 5-27

微服务将会通过 Zuul 和 Eureka 服务器进行负载均衡。

为其他服务完成 Zuul

为了完成我们的练习,为所有的微服务添加 API 网关项目(命名为*-apigateway)。按照下面的步骤进行。

- (1) 为每个服务新建属性文件。
- (2) Application.properties 重命名为 bootstrap.properties, 添加需要的配置。
- (3) 在每个*-apigateway 中为 Application.java 添加@EnableZuulProxy和@EnableDiscoveryClient 注解。
- (4) 如果有需要,可以修改默认的包名和文件名。

最后,我们得到了下列 API 网关项目:

- Chapter5.fares-apigateway。
- Chapter5.search-apigateway。
- Chapter5.checkin-apigateway。
- Chapter5.book-apigateway。

反应式微服务流

Spring Cloud 流提供了对信息传送基础设施的抽象化。底层的通信实现可以靠 RabbitMQ、Redis 或者 Kafka。Spring Cloud Stream 为收发消息提供了声明式的方法。

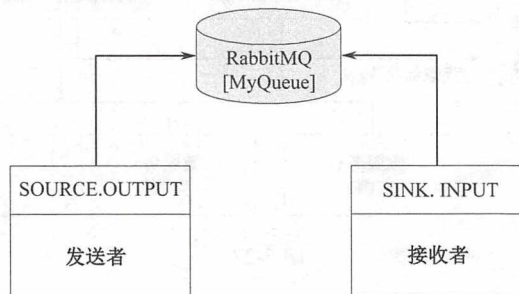


图 5-28

如图 5-28 所示，Cloud Stream 工作基于 source 和 sink 的概念。Source 代表消息的发送者，sink 则代表消息的接收者。

图 5-28 中的例子，发送者定义了一个称为 SOURCE.OUTPUT 的逻辑队列，负责发送消息。接收者定义了一个称为 SINK.INPUT 的逻辑队列，负责接收消息。OUTPUT 到 INPUT 的物理连接通过上面的架构进行管理。在这种情况下，SOURCE.OUTPUT 和 SINK.INPUT 都连接到同一个物理队列——RabbitMQ 实现的 MyQueue。

Spring Cloud 提供了在一个应用中使用多个消息提供者的灵活性，例如从 kafka 的输入流连接到 Redis 的输出流而不需要管理复杂性。Spring Cloud Stream 是基于消息集成的基础。Cloud Stream Modules 子项目是另一个提供了许多通道实现的 Spring Cloud 库。

下一步，用 Cloud Stream 重构微服务间消息通信。如下图所示，我们在 Search 微服务下定义了连接到 InventoryQ 的 SearchLink。Booking 将会定义一个 BookingSource 用来发送库存变化消息到 InventoryQ。类似地，Check-in 定义了

CheckinSource 用来发送签到消息。Booking 定义了一个 sink，BookingSink 用来接收消息。BookingSink 和 CheckinSource 都要连接到 CheckInQ。

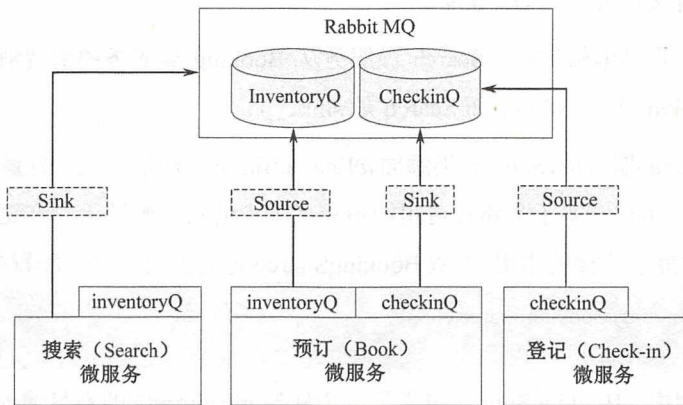


图 5-29

如图 5-29 所示，在这个例子中，我们使用 RabbitMQ 作为消息代理：

(1) 在 Booking、Ckeck-in 和 Search 3 个使用消息的模块中添加下面的 maven 依赖：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit
</artifactId>
</dependency>
```

(2) 在 booking-service.properties 中添加下面的两个属性，这些属性将逻辑队列 inventoryQ 绑定到物理队列 inventoryQ，同理，将逻辑队列 checkinQ 绑定到物理 checkinQ：

```
spring.cloud.stream.bindings.inventoryQ.destination=inventoryQ
spring.cloud.stream.bindings.checkInQ.destination=checkInQ
```

(3) 在 search-service.properties 中添加下列属性。这个属性将逻辑队列 inventoryQ 绑定到物理 inventoryQ：

```
spring.cloud.stream.bindings.inventoryQ.destination=inventory
```

(4) 在 checkin-service.properties 中添加下面的属性，将逻辑队列 checkinQ 绑定

到物理 checkInQ:

```
spring.cloud.stream.bindings.checkInQ.destination=checkInQ
```

(5) 所有文件提交到 Git 仓库。

(6) 下一步是编辑代码。Search 微服务从 Booking 微服务中消费消息。在这种情况下，Booking 就是 source 而 Search 是 sink。

给 Booking 服务的 Sender 类添加@EnableBinding 注解。这个注解的作用是让 Cloud Stream 工作在基于类路径可用的消息代理库的自动配置上。在这个例子中，RabbitMQ 发挥了这样的作用。参数 BookingSource 定义了用于这个配置的逻辑通道：

```
@EnableBinding(BookingSource.class)
public class Sender {
```

(7) 本例中，BookingSource 定义了一个称为 inventoryQ 的消息通道，如我们的配置所示，物理绑定到 RabbitMQ 的 inventoryQ。BookingSource 使用@Output 注解，来声明这个消息是输出型，从模块内部向外发送。这个信息用于消息通道的自动配置：

```
interface BookingSource {
    public static String InventoryQ="inventoryQ";
    @Output("inventoryQ")
    public MessageChannel inventoryQ();
}
```

(8) 如果服务只有一个 source 和一个 sink，我们使用 Spring Cloud Stream 默认的 Source 类，而不是自定义类：

```
public interface Source {
    @Output("output")
    MessageChannel output();
}
```

(9) 在发送器定义一个基于 BookingSource 的消息通道。下面的代码会注入一个名为 inventory 的输出消息通道，它已经在 BookingSource 中配置：

```
@Output(BookingSource.InventoryQ)
@Autowired
private MessageChannel;
```

(10) 重新实现 BookingSender 中的发送消息方法：


```
public void send(Object message){
    messageChannel.send(MessageBuilder.withPayload(message).build());
}
```

(11) 和我们之前在 Booking 服务中的做法一样，在 SearchReceiver 类中加入下面的代码：

```
@EnableBinding(SearchSink.class)
public class Receiver {
```

(12) 这样，SearchSink 接口如下所示。这会定义它所连接的逻辑 sink 队列。消息通道中加入 @Input 注解，表示这个消息通道是用于接收消息的：

```
interface SearchSink {
    public static String INVENTORYQ="inventoryQ";
    @Input("inventoryQ")
    public MessageChannel inventoryQ();
}
```

(13) 修改 Search 服务，让它可以接收消息：

```
@ServiceActivator(inputChannel = SearchSink.INVENTORYQ)
public void accept(Map<String,Object> fare){
    searchComponent.updateInventory((String)fare.
        get("FLIGHT_NUMBER"),(String)fare.
        get("FLIGHT_DATE"),(int)fare.
        get("NEW_INVENTORY"));
}
```

(14) 我们仍然需要 RabbitMQ 的配置来连接到消息代理：

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

(15) 启动所有服务，运行整个 Web 项目。如果一切顺利，Web 项目将会成功地执行 Search、Booking 和 Check-in 功能。可以在浏览器中打开 <http://localhost:8001> 进行测试。

BrownFeild PSS 架构总结

图 5-30 展示了我们创建的整个架构，包括 Config 服务器、Eureka、Feign、Zuul 及 Cloud Stream。这个架构也包含了所有组件的高可用。假设客户端使用 Eureka 客户端库。

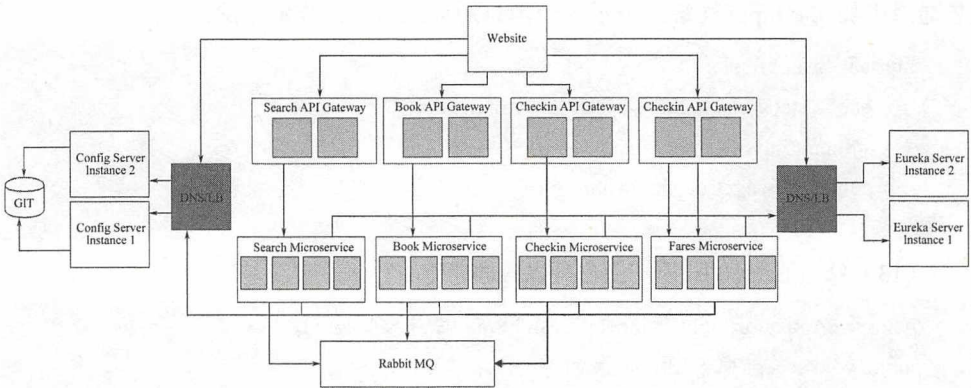


图 5-30

各个项目及它们各自监听的端口如图 5-31 所示。

Microservice	Projects	Port
Book microservice	chapter5.book	8060 to 8064
Check-in microservice	chapter5.checkin	8070 to 8074
Fare microservice	chapterb.fares	8080 to 8084
Search microservice	chapter5.search	8090 to 8094
Website client	chapter5.website	8001
Spring Cloud Config selver	chapter5.configserver	8888/8889
Spring Cloud Eureka server	chapter5.eurekaserver	8761/8762
Book API gateway	chapter5.book-apigateway	8095 to 8099
Check-in API gateway	chapter5.checkin-apigateway	8075 to 8079
Fares API gateway	chapter5.fares-apigateway	8085 to 8089
Search API gateway	chapter5.search-apigateway	8065 to 8069

图 5-31

按下列步骤完成最后的运行：

- (1) 运行 RabbitMQ。

(2) 在根目录的 pom.xml 构建整个项目：

```
mvn -Dmaven.test.skip=true clean install
```

(3) 从各个项目的文件夹中运行下列项目。记得在启动下一个服务之前等待 40~50s。这样做可以保证每个独立的服务注册并且可用。

```
java -jar target/fares-1.0.jar
java -jar target/search-1.0.jar
java -jar target/checkin-1.0.jar
java -jar target/book-1.0.jar
java -jar target/fares-apigateway-1.0.jar
java -jar target/search-apigateway-1.0.jar
java -jar target/checkin-apigateway-1.0.jar
java -jar target/book-apigateway-1.0.jar
java -jar target/website-1.0.jar
```

(4) 打开浏览器，输入 <http://localhost:8001>，按照第 4 章中“运行和测试项目”小节的步骤做。

总结

本章中，你学习了如何使用 Spring Cloud 项目来扩（缩）容一个十二因素 Spring Boot 微服务。学到的知识应用在我们前面几章开发的 BrownField 航空 PSS 微服务。

然后，我们研究了用于微服务配置统一中心的 Spring Config 服务器，并且介绍了如何让它高可用。我们也讨论了使用 Feign 进行声明式的服务调用，检查了 Ribbon 和 Eureka 在负载均衡、服务注册和发现中的作用。通过 Zuul 实现了 API 网关。最后，我们使用 Spring Cloud Stream 总结了响应式风格的微服务。

BrownField 航空 PSS 微服务可以在互联网规模上部署。其他的 Spring Cloud 组件（如 Hyterix、Sleuth 等）将会在第 7 章（日志记录和监控微服务）中做讨论。下一章我们会重点介绍自动化扩（缩）容功能，从而扩展 BrownField 航空 PSS 的实现。

第 6 章

自动化扩（缩）容微服务



Spring Cloud 提供了微服务部署规模方面的关键支持。为了最大化利用云计算环境的功能，微服务实例应该能够基于流量模式自动扩（缩）容。

这一章会详细介绍如何有效利用从 Spring Boot 微服务收集的 Actuator 数据，使微服务弹性扩容和缩容，从而通过实现简单生命周期管理器来控制部署拓扑。

学习完本章后，您会了解到下面的知识点：

- 自动化扩（缩）容的概念和实现方法。
- 在微服务上下文中生命周期管理器的重要性和功能。
- 检查自定义生命周期管理器，从而实现自动化扩（缩）容。
- 从 Spring Boot Actuator 收集数据，并用这些数据控制和整形流量。

回顾微服务功能模型

本章会介绍第 3 章中讨论过的微服务功能模型中的 Application Lifecycle Management 功能，如图 6-1 高亮所示。

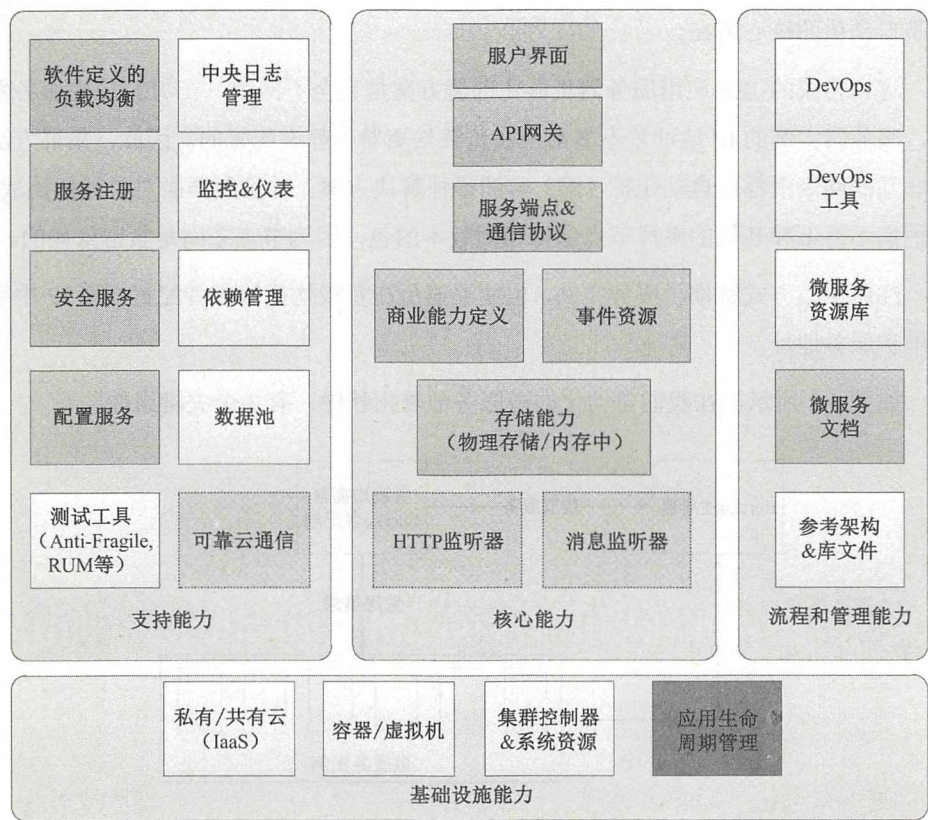


图 6-1

本章中我们会得到一个基础版本的生命周期管理器，后面几章中将会做进一步改进。

用 Spring Cloud 扩（缩）容微服务

在第 5 章中，我们学会了如何使用 Spring Cloud 组件扩（缩）容 Spring Boot 微服务。Spring Cloud 的两个关键概念是服务的自我注册和自我发现。这两个功能实现了自动化的微服务部署。有了自我注册，一旦实例可以接受流量，微服务可以通过注册服务元数据到服务注册中心来自动广播其可用性。一旦微服务注册成功，消费者就可以通过使用注册中心服务发现服务实例，从而消费最新注册的服务。注册中

心是自动化的核心所在。

这和传统的 JEE 应用服务器集群化部署方案是完全不同的。在 JEE 应用服务器中，服务器实例的 IP 地址差不多都是在负载均衡器中静态配置的。因此，集群方法并非互联网规模部署自动化扩（缩）容的最佳解决方案。并且集群面对了其他挑战，如他们不得不和其他的集群节点保持相同版本的包，因为节点之间是紧密依赖的。

注册中心方式解耦了服务实例，也能够避免在负载均衡器或者配置虚拟 IP 中手动维护服务地址。

如图 6-2 所示，在我们自动化的微服务部署拓扑中，有 3 个关键组件：

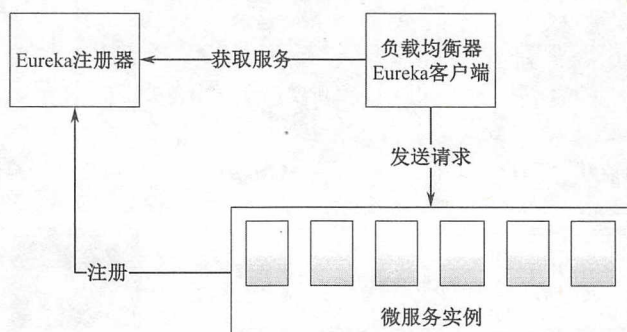


图 6-2

- Eureka 是注册中心组件，用于微服务的注册和发现。消费者和生产者都使用 REST API 来连接注册中心。注册中心也持有服务的元数据，如服务身份、主机、端口、健康状况等。
- Eureka 客户端和 Ribbon 客户端一起，提供客户端的动态负载均衡。消费者使用 Eureka 客户端来查找 Eureka 服务器中可用的目标服务实例。Ribbon 客户端使用查询到的服务器列表在可用的微服务实例之间进行负载均衡。类似地，如果服务实例不可用，这些实例会从 Eureka 注册中心中被剔除掉。负载均衡器自动响应动态拓扑的改变。
- 第 3 个组件是使用 Spring Boot 开发的微服务实例。

然而，这种方式有一个难题，当需要传统的微服务实例时，需要手动创建一个新的实例。理想情况下，微服务实例的启动和停止也需要自动化实现。

例如，当我们需要添加另一个微服务实例来处理流量的增加或者负载过度，管理者需要手动创建一个新的实例。并且当 Search 服务的实例空闲时，需要手动从服务中剔除，从而保证整个框架的最佳利用。在按照使用量付费的云环境中，这一点尤其重要。

理解自动化扩（缩）容的概念

自动化扩（缩）容是自动基于资源使用对实例进行复制扩容，从而符合 SLA。系统自动发现流量增长，自动增加额外的实例，并让它们可用于流量处理。类似地，当流量下降，系统自动发现并减少实例数量，让活跃的实例继续留在服务列表中。

如图 6-3 所示，自动化缩容/扩容是通过使用一系列备用机实现的。

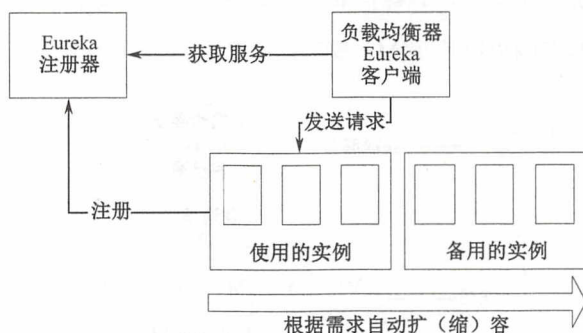


图 6-3

因为许多云订阅基于即付即用模型，当针对于云部署的时候，这一点尤其关键。这个方法被称为弹性配置，也称为动态资源配置和解除配置。自动化扩（缩）容是专门针对有变化的流量模式的微服务。例如，一个 Accounting 服务在月底或者年底的时候可能会有很高的流量，没必要为了季节性的业务而一直保持超负荷的配置实例。

在自动化扩（缩）容的过程中，会有充足的空闲实例在一个资源池中。当有大业务的请求，实例会从资源池移到活跃状态从而满足需求。这些实例不预先配置任何特定的微服务或者预先打包任何微服务包，当有需要的时候，Spring Boot 微服务包会按需从一个 artifact 库（如 Nexus 或者 Artifactory）中下载并部署起来。

自动化扩（缩）容的好处

实行自动化扩（缩）容有很多好处。在传统的部署中，管理者为应用备用一系列服务器。有了自动化扩（缩）容，就没必要这样预分配了。预先设定的服务器分配可能导致某些服务器未被充分利用。这种情况下，就算有些服务一直在争夺额外的资源，周围的空闲服务器仍不能被利用。

当系统中有上百个微服务实例，预先分配固定数量的服务器到每个微服务下是不合算的。一个更好的方法是，为一组微服务预留一部分服务实例；但不预先分配或者标上某一个微服务的标签。基于需求，一组服务共享一系列可用资源。这样微服务就可以通过优化资源使用，从而动态地在可用的服务器间移动。

如图 6-4 所示，M1 微服务有 3 个实例，M2 和 M3 则各有一个在运行。还有一个服务器未被分配。根据需求，未分配的服务器可以被用于 M1、M2 或者 M3 中的任何一个。如果 M1 有更多的服务请求，就把未分配的服务器用于 M1。当服务使用率下降，这个服务实例可以得到释放并重新移回到池中。稍后，如果 M2 的需求上升，同样的服务器实例可以使用 M2 激活。

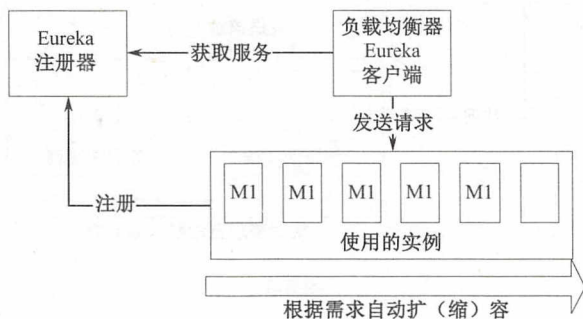


图 6-4

自动化扩（缩）容的一些关键好处在于：

- 有很高的可用性和容错性：因为有许多服务实例，即使其中一个失效，另一个实例也可以接替并继续为客户端服务。这种失效对于消费者而言是透明的。如果这个服务没有其他实例可用，自动化扩（缩）容会意识到这种情况并且安排另一个服务的实例。因为所有启动和关闭实例都是自动化的，服务的整体可用性会高于没有进行自动化扩（缩）容的系统。在没

有进行自动化扩（缩）容的系统中，需要手动添加或者删除服务实例，在大规模部署中将很难实现。

例如，假设 Booking 服务的两个实例在运行。如果流量出现了增加，通常情况下，已存在的实例会过载，整个服务就可能会堵塞，导致服务不可用。有自动化扩（缩）容的时候，一个新的 Booking 服务的实例会很快出现，从而均衡负载并保证服务的可用性。

- 增加了扩展性：自动化扩（缩）容的一个关键好处是水平伸缩性。它允许我们基于流量模式有选择地扩大或者缩小服务。
- 使用最优而且节约成本：在一个现收现付的订阅模型中，费用给予实际的资源使用。有了自动化扩（缩）容，实例会根据实际需求启动或者关闭。因此，资源得到了最优化利用，并且节约了成本。
- 可以给某些服务设置优先级：有了自动化扩（缩）容，我们可以让关键交易的优先级高于低价值交易。实现方式是将低价值交易的服务实例减少，把资源分配给高价值服务。这样做也能避免低价值服务占用过多使用资源从而导致资源紧张。

如图 6-5 的例子所示，Booking 和 Reports 服务运行着两个实例。我们假设 Booking 服务是一个创收服务，有着比 Reports 服务更高的价值。如果 Booking 服务需要更多资源，我们可以设置将一个 Report 服务释放掉，并将该服务器用于 Booking 服务。

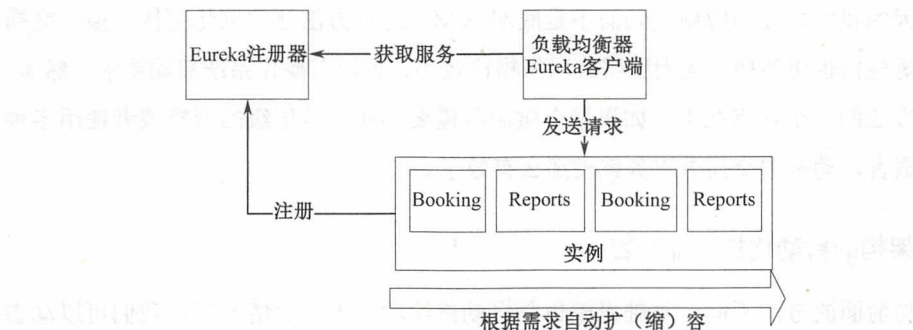


图 6-5

不同的自动化扩（缩）容模型

自动化扩（缩）容能够用于应用层级或者架构层级。简而言之，应用的自动化

扩（缩）容只是通过复制应用本身，架构的自动化扩（缩）容是复制整个虚拟机，包括应用本身。

应用的自动化扩（缩）容

这种场景下，扩（缩）容是通过复制微服务实现的，而不是底层架构（如虚拟机）。假设是有一个维护着 VM 或者物理基础设施的池。这些 VM 拥有依赖的基础映像，如 JRE。另一个假设是，微服务天生就是同性质。这就让在不同的服务中复用同一个虚拟或物理机成为可能。

如图 6-6 所示，在场景 A 中，VM3 用于服务 1；而场景 B 中，同一个 VM3 用于服务 2。这时，我们只是交换了应用库，而不是底层架构。

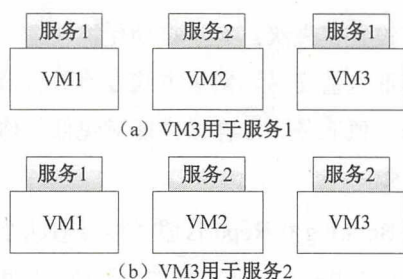


图 6-6

因为我们只处理应用代码而不是底层 VM，这种方法让实例化更快交换。交换过程更轻松也更便捷，因为应用代码规模比较小，也没有操作系统启动要求。然而，这种方法的一个缺点在于，如果某个微服务需要操作系统层级的调整或者使用多种编程语言，动态的交换微服务就没那么有效了。

架构的自动化扩（缩）容

与前面的方法不同，基础设施也是自动供给的。大多数情况下，我们可以动态地创建一个新的 VM，也可以按需删除它们。

如图 6-7 所示，预先创建一些服务实例的 VM 映像，当服务 1 有需求时，VM3 被移动到活跃状态。当服务 2 有需求的时候，VM4 被移动到活跃状态。

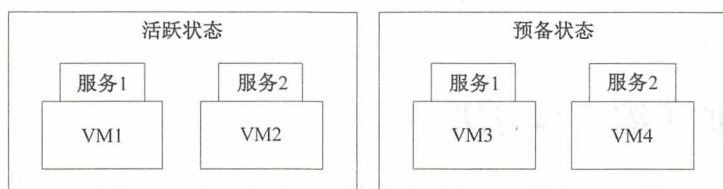


图 6-7

如果应用依赖于架构层级的参数和库（如操作系统），这种方式是很有效的。并且，如果微服务有多种编程语言时，这种方式更为有效。不足之处是，VM 映像很重，启动一个新的 VM 也需要一定时间。这时候倾向于选择如 Docker 的轻量级容器，而不是传统的重量级虚拟机。

在云端自动化扩（缩）容

弹性或者自动化扩（缩）容是大多数云服务提供商的基础功能之一。云服务提供商使用前面几章介绍的架构伸缩模式，它们主要基于机器集群。

例如，在 AWS 中，这些是基于引入一个新的有预定义 AMI 的 EC2 实例。AWS 借助自动化扩（缩）容组来提供自动化扩（缩）容支持。每一个组都有最多和最少数量的实例。AWS 保证实例的扩（缩）容在这个范围之内。在可预测流量模式的情况下，可以基于时间线来进行配置。AWS 也可以为应用提供自定义的自动化扩（缩）容策略。

Microsoft Azure 也支持基于资源（如 CPU、消息队列长度等）使用情况的自动化扩（缩）容，IBM Bluemix 支持如 CPU 使用率等资源的自动化扩（缩）容。

其他的 PaaS 平台，如 CloudBees 和 OpenShift，也支持 java 应用的自动化扩（缩）容。Pivotal Cloud Foundry 借助 Pivotal Autoscale 支持自动化扩（缩）容。扩（缩）容策略基于资源的使用情况，如 CPU 和内存阈值。

一些运行在云顶端的组件，提供细粒度控制来处理自动化扩（缩）容，如 Netflix Fenzo、Eucalyptus、Boxfuse 和 Mesosphere。

自动化扩（缩）容方法

自动化扩（缩）容的实现，主要是考虑不同的参数和阈值。本节中，我们会讨论典型的业务场景中什么情况下扩（缩）容的具体方法和策略。

根据资源限制扩（缩）容

这个方法基于监控机制得到的实时服务指标。通常来说，根据资源决定规模的方法，主要是考虑机器的 CPU、内存、硬盘。也可以通过服务实例自身得到的数据，如堆内存使用。

一个典型的策略是，当 CPU 使用率超过 60%，就建立另一个实例。类似地，如果堆的大小超过某个确定的阈值，我们也可以添加一个新的实例。同理，当资源的使用率低于某个设定的阈值，就应该通过逐步关闭服务器来缩减计算容量，如图 6-8 所示。

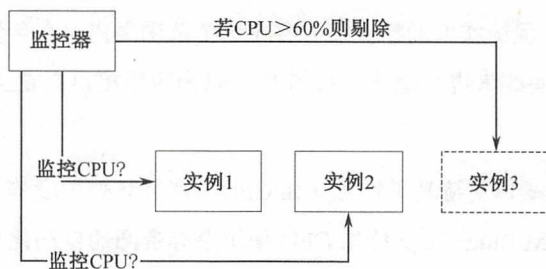


图 6-8

在典型的生产场景下，并不是第一次突破阈值就需要额外创建服务。正确的方式是，定义一个滑动窗口或者等待期。

下面是一些例子：

- 一个响应滑动窗口的例子是，在一个 60s 的采样窗口，如果某个特定交易的六成响应时间都高于设定的阈值，就增加服务实例。

- 对于 CPU 滑动窗口，在一个 5min 的滑动窗口中，如果 CPU 使用率一直高于 70%，就创建新的实例。
- 对于异常滑动窗口，如果 60s 滑动窗口的八成交易，或者连续 10 个执行都导致特定的系统异常，如因为线程池紧张而导致连接超时，就创建一个新的实例。

在很多情况下，我们设置的阈值会比期望值低一些。例如，设置 CPU 使用率阈值为 60% 而不是 80%，这样系统在停止相应之前就有足够的时间创建新的实例。类似地，当缩容时，我们设定的阈值也会比期望值低。例如，设置阈值为 40% 而不是 60% 的时候进行缩容。这样做可以给我们一个冷却期，不至于因为关闭实例而导致资源争夺。

基于资源的扩（缩）容，也可以应用于服务层级的参数，如服务的吞吐量、延迟、应用线程池、连接池等。这些也可以在应用层级，如内部服务实例中销售订单的数量。

在特定时间扩（缩）容

基于时间的扩（缩）容是指在特定的日期、月份或者年限，处理季节性或者业务峰值。例如，一些交易可能在工作时间出现峰值，而在工作时间之外，交易量却很低。在这种情况下，就应该在一天中，根据需求对服务进行扩（缩）容，在非工作时间自动进行缩容，如图 6-9 所示。



图 6-9

世界上的许多机场限制夜晚降落。相比于白天，夜晚登机的乘客数量要少得多。因此，夜晚关闭服务实例的数量是高效而且节约成本的。

基于消息队列长度进行扩（缩）容

当微服务基于异步消息的时候，这样做尤其有效。当消息队列的长度超过某个特定限制时，就会自动添加新的消费者，如图 6-10 所示。

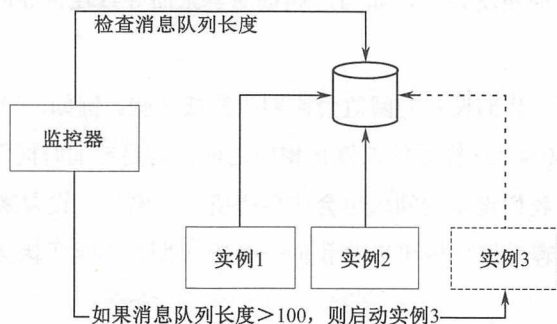


图 6-10

这种方法基于竞争消费模式，实例池用来消费消息。基于消息阈值，添加新实例去消费额外的消息。

基于业务参数进行扩（缩）容

这种情况下，添加实例主要基于特定的业务参数。例如，在处理销售结算之前，添加新的实例。一旦监控服务接收到预先配置的业务时间（如销售收盘前一小时），就会创建一个新的实例用来处理大量交易。这样做可以给予业务规则，对扩（缩）容进行细粒度控制，如图 6-11 所示。

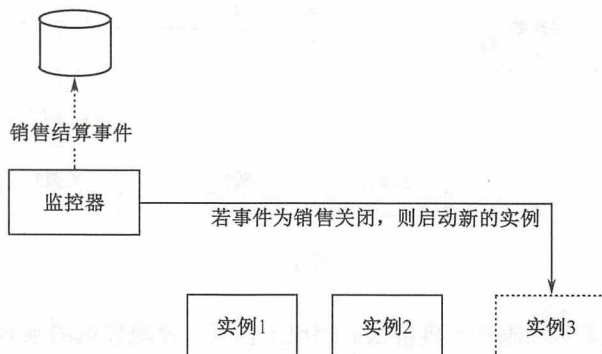


图 6-11

预测性的自动化扩（缩）容

预测式扩（缩）容是新的自动扩（缩）容模式，不同于传统的基于指标的实时自动化扩（缩）容。一个预测引擎会接受很多输入（如历史信息、当前趋势等）来预测可能的流量模式。基于这些预测，系统可以进行自动化扩（缩）容。这种方式最大的好处在于，可以避免硬编码规则和时间窗口。系统可以自动预测这样的时间窗口。在更加复杂的部署当中，预测分析可能使用认知计算机机制来预测何时进行自动化扩（缩）容。

一旦出现突然的流量峰值，传统的自动化扩（缩）容可能会失效。在自动化扩（缩）容组件根据情况作出响应之前，流量峰值可能已经摧毁了系统。预测系统可以理解这些场景并且在实际发生之前作出预测。例如，在系统停机之后，大量请求将涌入需要处理。

Netflix Scryer 就是这样的系统，可以提前预测资源需求。

对 BrownField PSS 微服务自动化扩（缩）容

本节中，我们会探讨如何增强第5章中开发的微服务，使其可以自动化扩（缩）容。我们需要一个组件来监控特定的性能指标，并且触发自动化扩（缩）容，这个组件称为生命周期管理器。

服务生命周期管理器，或者称为应用生命周期管理器，负责发现扩（缩）容的需求，并且根据情况调整实例数量，它负责动态启动或者关闭实例。

本节中，我们会创建一个简单的自动化扩（缩）容系统来理解基础概念，后面几章中我们会进一步探讨。

自动化扩（缩）容系统的必备功能

典型的自动化扩（缩）容系统应该具有图6-12中的功能。

微服务中用于自动化扩（缩）容的组件包括：

- 微服务：这是一系列正在运行的微服务实例，一直发送健康状况和指标信息。这些服务为指标收集暴露执行端点，在上图中，它们分别是微服务1~4。

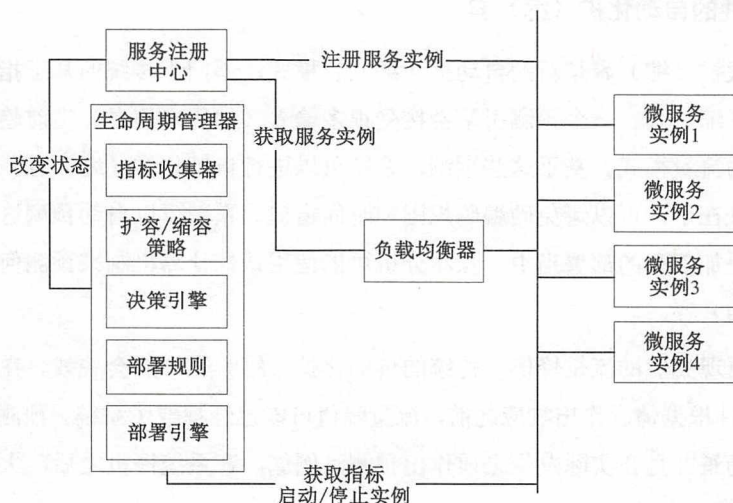


图 6-12

- 服务注册中心：服务注册中心跟踪所有微服务及其健康状态、元数据和 URL。
- 负载均衡器：这个客户端负载均衡器从服务注册中心得到最新的可用服务实例信息。
- 生命周期管理器：负责自动扩（缩）容，由下列子组件组成。
 - 指标收集器：指标收集器单元负责从所有服务实例搜集指标数据。生命周期管理器会汇总这些指标，并运行一个滑动时间窗口。指标可以是架构层级的指标，如 CPU 使用率，或者是应用层级的指标，如每分钟的交易所数。
 - 扩容/缩容策略：扩容/缩容策略是一系列规定什么时候扩容、什么时候缩容的策略。例如，5min 的滑动窗口，九成的 CPU 使用率超过 60%。
 - 决策引擎：决策引擎负责基于汇总的指标和扩（缩）容策略，来决定扩容还是缩容。
 - 部署规则：部署引擎使用部署规则来决定，在部署服务的时候要考虑哪些参数。例如，一个服务部署限制规定，一个服务的实例必须分布多个可用区域或者至少分配 4GB 内存。

- 部署引擎：部署引擎，基于决策引擎的决策，可以启动或是停止微服务实例，或者通过改变服务的健康状态来更新决策中心。例如，设置健康状态为“停用”来暂时停止一个服务。

使用 Spring Boot 实现自定义生命周期管理器

本节中的生命周期管理器是理解自动化扩（缩）容功能的最基础实现。后面几章中，我们会用容器和集群管理解决方案来深化该实现。Absible、Marathon 和 Kubernetes 是实现该功能的一些有效工具。

本节中，针对第5章中开发的服务，我们会使用 Spring Boot 实现应用层级的自动化扩（缩）容组件。

理解部署拓扑

图 6-13 展示了 BrownField PSS 微服务的采样部署拓扑：

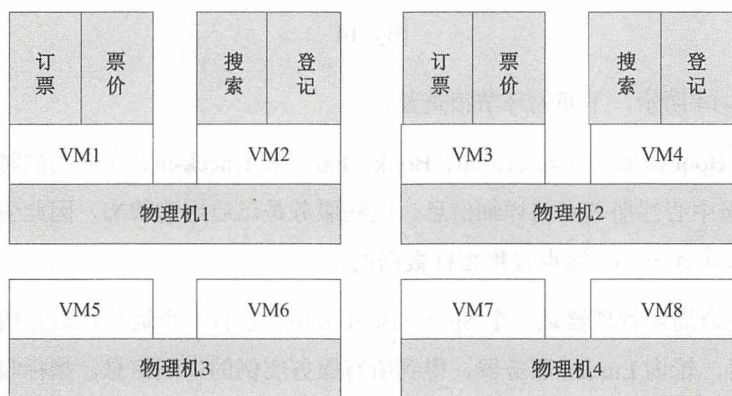


图 6-13

如图 6-13 所示，有 4 个物理机，8 个 VM。每个物理机有 2 个 VM，每个 VM 可以运行 2 个 Spring Boot 实例，假设所有服务的资源需求都是相同的。

VM1 到 VM4 已经被激活，可以处理流量。VM5 到 VM8 用作预留 VM，来处理扩（缩）容。VM5 和 VM6 可以用于任何微服务，也可以基于扩（缩）容需求，在不同微服务之间切换。冗余的服务使用不同物理机创建的 VM 来改善容错。

我们的目标是，当流量增加时，使用 VM5~VM8 这 4 个 VM，实现任何服务的横向扩展。当没有足够的负载时，收缩微服务，我们的架构如下所示。

理解执行流

看图 6-14 所示的流程图。

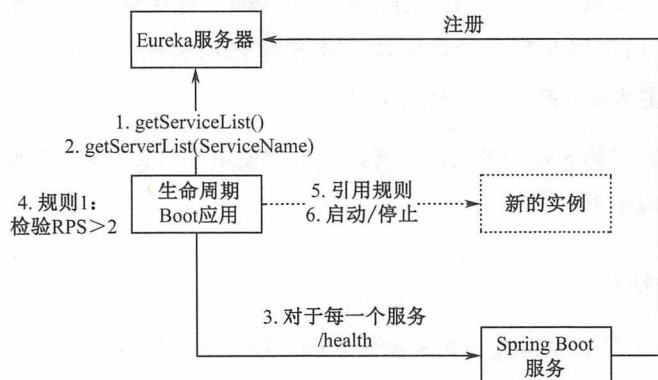


图 6-14

如图 6-14 所示，下面的环节很重要：

Spring Boot 微服务（如 Search、Book、Fares 和 Check-in）在启动的时候自动在 Eureka 注册中心注册端点的详细信息。这些服务是已启用的状态，因此生命周期管理器是可以从 Actuator 端点收集指标数据的。

- 生命周期管理器是一个 Spring Boot 应用。它有一个运行在后台的指标收集器，轮询 Eureka 服务器，得到所有服务实例的详细信息。指标收集器会激发注册在 Eureka 注册中心的每个微服务的执行端，从而得到健康状况和指标信息。在真实的生产环境中，数据收集的订阅方法更有效。
- 有了收集到的指标信息，生命周期管理器执行一系列策略，并决定对实例进行扩容还是缩容。这些决策会在某个 VM 创建某种新的服务实例，或者关闭特定的实例。
- 关闭实例的时候，使用执行端连接到服务器，调用关闭服务来优雅地关闭实例。

- 新建一个实例的时候，生命周期管理器的部署引擎使用扩（缩）容规则，决定在哪里启动新建的实例，以及启动实例的时候使用什么参数。然后使用 SSH 连接到各自的 VM。一旦连接成功，通过传递需要的限制作为参数，执行一个预安装的脚本（或者作为执行的一部分，传递脚本）。脚本从存储着生产库的中心 Nexus 库取得应用库，并将其初始化为一个 Spring Boot 应用。生命周期管理器参数化端口号。SSH 需要在目标机上激活。

在本例中，我们会使用 TPM（每分钟交易数）或者 RPM（每分钟请求数）作为采样参数来进行决策的制定。如果 Search 服务超过 10TPM，就会新建一个 Search 服务的实例。类似地，如果 TPM 低于 2，就会关闭一个实例，释放回到池中。

当启动新的实例时，需要应用到下面的策略：

- 任何一点的服务实例数量应该介于 1~4 之间，也就意味着，至少有一个服务实例在运行中。
- 对于缩放组的定义是，一个新的服务实例被创建在不同物理机上的 VM。这样可以保证服务运行在多个不同的物理机上。

上面的策略可以进一步深化。理想状态下，生命周期管理器通过 REST API 或者 Groovy 脚本，提供这些规则的自定义选项。

生命周期管理器代码的演练

接下来我们看看，一个简单的生命周期管理器是如何实现的。本节会演练一下源码，来理解生命周期管理器的不同组建。

完整的源代码在 Chapter 6 项目的代码文件中。将 `chapter5.configserver`、`chapter5.eureka-server`、`chapter5.search` 和 `chapter5.searchapigateway` 复制到该文件夹下，并且分别重命名为 `chapter6.*`。

按照下列步骤来实现一个生命周期管理器：

(1) 新建一个 Spring Boot 应用，并命名为 `chapter6.lifecyclemanager`，项目结构如图 6-15 所示。



图 6-15

本例的流程图如图 6-16 所示。

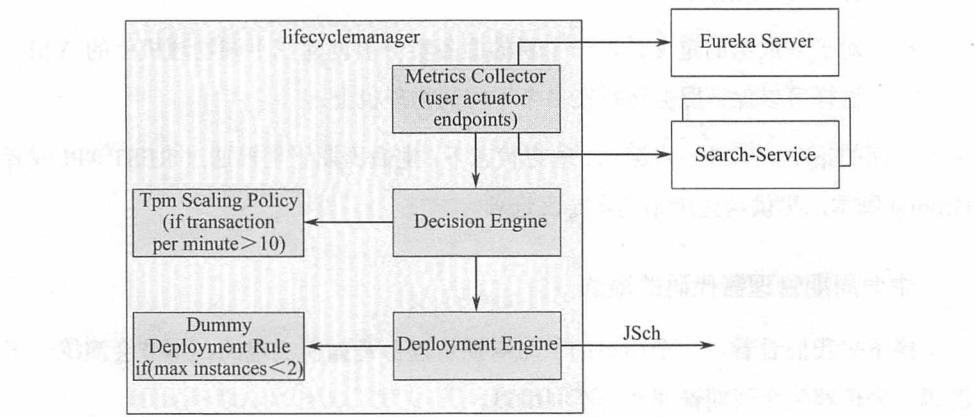


图 6-16

图中的组件会详细介绍。

(2) 新建一个 MetricsCollector 类，如下所示。在 Spring Boot 应用启动的时候，CommandLineRunner 会调用这个方法：

```
public void start(){
    while(true){
        eurekaClient.getServices().forEach(service -> { System.out.println("discovered service"
+ service);
```



```

        Map metrics = restTemplate.getForObject("http://" + service + "/"
        metrics", Map.class);
        decisionEngine.execute(service, metrics);
    });
}
}

```

上面的方法查找注册在 Eureka 服务器的服务并得到所有实例。现实中,实例将指标发布到一个公共区域进行指标聚集而非轮询。

(3) 下面的 DecisionEngine 代码接受指标,并用伸缩策略来决定这个服务应该扩容还是缩容。

```

public boolean execute(String serviceId, Map metrics){
    if(scalingPolicies.getPolicy(serviceId).execute(serviceId, metrics)){
        return deploymentEngine.scaleUp(deploymentRules.
        getDeploymentRules(serviceId), serviceId);
    }
    return false;
}

```

(4) 基于服务 ID,服务的指标信息将会存储起来。这时在 TpmScalingPolicy 中实现最小 TPM 的伸缩策略,如下所示:

```

public class TpmScalingPolicy implements ScalingPolicy {
    public boolean execute(String serviceId, Map metrics){
        if(metrics.containsKey("gauge.servo.tpm")){
            Double tpm = (Double) metrics.get("gauge.servo.tpm");
            System.out.println("gauge.servo.tpm " + tpm);
            return (tpm > 10);
        }
        return false;
    }
}

```

(5) 如果该策略返回值为 true, DecisionEngine 会调用 DeploymentEngine,从而创建一个新的实例。Deployment Engine 使用 DeploymentRules 来决定如何实现扩(缩)容。规则可以强制设定最大或者最小的实例数量,新的实例在哪里启动,以及新实例需要的资源等。DummyDeploymentRule 的最大实例数不能超过 2 个。

(6) 从 JCraft 到 SSH 再到目标服务器并且启动服务, Deployment Engine 使用 JSCH(javasecure Channel)库。这需要在 maven 中添加下面的依赖:

```
<dependency>
  <groupId>com.jcraft</groupId>
  <artifactId>jsch</artifactId>
  <version>0.1.53</version>
</dependency>
```

(7) 当前的 SSH 实现是很简单的, 后面几章我们会加深。在本例中, Deployment Engine 在目标机上发送下面的请求到 SSH 库。

```
String command ="java -jar -Dserver.port=8091 ./work/codebox/
chapter6/chapter6.search/target/search-1.0.jar";
```

也有目标机需要使用带 Nexes CLI 的 Linux 脚本或使用 curl 的时候才会与 Nexus 一起集成的情况, 本例中我们不会探究 Nexus。

(8) 下一步是, 修改 Search 微服务, 为 TPM 暴露新的测量值。我们必须修改之前开发的微服务, 提交这个额外的指标。

本章中我们只会检测 Search。但是为了完成整个项目, 所有的服务都需要更新。为了得到 gauge.servo.tpm 指标, 我们需要给所有微服务添加 TPMCounter。

下面的代码在 1min 滑动窗口中, 记录交易数。

```
class TPMCounter {
    LongAdder count;
    Calendar expiry = null;
    TPMCounter(){
        reset();
    }
    void reset (){
        count = new LongAdder();
        expiry = Calendar.getInstance();
        expiry.add(Calendar.MINUTE, 1);
    }

    boolean isExpired(){
        return Calendar.getInstance().after(expiry);
    }
}
```



```

    }
    void increment(){
        if(isExpired()){
            reset();
        }
        count.increment();
    }
}

```

(9) 在 SearchController 中设置下面的 TPM 值:

```

class SearchRestController {
    TPMCounter tpm = new TPMCounter();
    @Autowired
    GaugeService gaugeService;
    //other code
}

```

(10) 下面的代码来自 SearchRestController 的 search 方法, 提交 TPM 值作为测量值到 actuator。

```

tpm.increment();
gaugeService.submit("tpm", tpm.count.intValue());

```

运行生命周期管理器

按照下面的步骤, 运行我们之前开发的生命周期管理器。

(1) 修改 DeploymentEngine.java, 更新密码, 如下所示, 这样做是为了 SSH 的连接。

```

session.setPassword("rajeshrv");

```

(2) 通过下面的代码, 运行根目录(第6章)的 Maven, 从而创建整个项目。

```

mvn -Dmaven.test.skip=true clean install

```

(3) 运行 RabbitMQ。

```

./rabbitmq-server

```

(4) 确保 Config 服务器指向正确的配置库。我们需要为生命周期管理器添加属性文件。

(5) 从各自的项目文件中, 运行下面的命令。

```

java -jar target/config-server-0.0.1-SNAPSHOT.jar
java -jar target/eureka-server-0.0.1-SNAPSHOT.jar

```

```
java -jar target/lifecycle-manager-0.0.1-SNAPSHOT.jar
java -jar target/search-1.0.jar
java -jar target/search-apigateway-1.0.jar
java -jar target/website-1.0.jar
```

- (6) 一旦所有服务器启动，打开浏览器窗口，输入<http://localhost:8001>。
- (7) 1min 内，依次执行航班搜索 11 次。这会触发决策引擎来实例化一个 Search 微服务的实例。
- (8) 打开 Eureka 控制台 (<http://localhost:8761>)，找第二个 SEARCH-SERVICE。一旦服务器启动成功，实例就会出现图 6-17 所示界面。

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
LIFECYCLE-MANAGER-SERVICE	n/a (1) (1)		UP (1) - 192.168.0.106:lifecycle-manager-service:9090
SEARCH-APIGATEWAY	n/a (1) (1)		UP (1) - 192.168.0.106:search-apigateway:8095
SEARCH-SERVICE	n/a (2) (2)		UP (2) - 192.168.0.106:search-service:8091, 192.168.0.106:search-service:8090
TEST-CLIENT	n/a (1) (1)		UP (1) - 192.168.0.106:test-client:8001

图 6-17

总结

在本章中，我们学习了部署大规模微服务时自动化扩（缩）容的重要性。我们也学习了自动化扩（缩）容的概念、不同的模型和实现自动化扩（缩）容的方法，如基于事件、基于资源、基于队列长度以及可预测方法等。我们回顾了生命周期管理器在微服务中的作用和功能。最后，我们通过回顾 BrownFieild PSS 微服务如何用简单的自定义生命周期管理器的采样实例来结束本章内容。

面对大规模微服务时，自动化扩（缩）容是一个重要的支持性功能。我们会在第 9 章中实现一个更成熟的生命周期管理器。

第 7 章会研究日志和监控功能，它们对于成功的微服务部署是不可或缺的。

第 7 章

日志记录和监控微服务



由于互联网级微服务部署具有分布式的特征，对单个微服务进行记录和监控具有很大的挑战。很难通过不同微服务所产生的日志来追踪到端到端事务。就像单体应用那样，没有一个单一管理平台来监控微服务。

本章会阐述记录和监控微服务部署的必要性和重要性。本章还会进一步介绍一系列潜在的架构和技术来解决日志记录和监控过程中所具有的挑战和困难。

通过本章，您可以学到以下知识：

- 使用不同的选项、工具和技术进行日志管理。
- 使用 Spring Cloud Sleuth 来追踪微服务。
- 使用不同工具对微服务进行端到端监控。
- 使用 Spring Cloud Hystrix 和 Turbine 进行熔断监控。
- 使用数据湖泊来进行业务数据分析。

回顾微服务能力模型

在本章中，我们会探讨在第 3 章“微服务概念的应用”中提到的微服务能力模型中的以下几种微服务的能力，如图 7-1 高亮所示。

- 中央日志管理。
- 监控和仪表盘。
- 依赖管理（监控和仪表盘的一部分）。
- 数据湖泊。

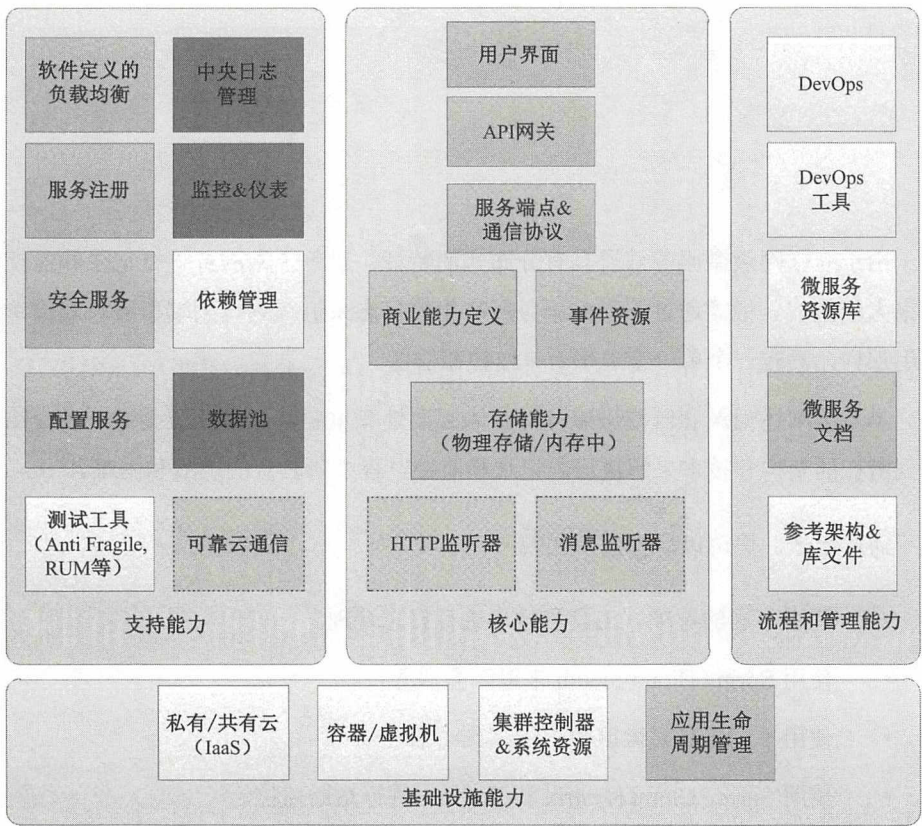


图 7-1

理解日志管理的挑战

日志就是运行的进程所产生的事件流。对于传统的 JEE 应用，有许多框架和库可以用来输出日志。Java Logging (JUL) 是 Java 自带的日志库。Log4j、Logback 及 SLF4J 是其他一些流行的日志框架。这些框架同时支持 UDP 和 TCP 协议来传输日志。应用把日志输出到控制台或文件系统中。文件回收技术通常会被使用，以防止日志文件占满了所有磁盘空间。

为了避免磁盘 IO 所产生的高开销，日志管理中一个好的做法是在生产中关闭大多数的日志入口。硬盘 IO 不仅会减慢应用的运行速率，还会对可扩展性带来极大的影响。把日志写到磁盘中还需要很大容量的磁盘。磁盘数据溢出这样的场景甚至会导致应用宕机。日志框架提供在运行时限制哪些需要打印、哪些不需要打印的选项。大多数这些框架对日志管理提供细粒度的控制。它们还提供在运行时更改这些配置的选项。

另外，日志可能包含重要的信息，如果对其进行恰当的分析会带来很高的价值。因此，限制日志的入口本质上限制了我们理解应用行为的能力。

当从传统部署转变到云部署，应用不再拘泥于一个特定的、已定义的机器。虚拟机和容器并非带有应用的硬件。用来部署的机器会不停地发生改变。而且，类似 Docker 的容器只是临时的。这也就意味着，我们不能完全依赖于硬盘来保持数据。一旦容器停止运行或者重启，保存在磁盘上的日志就会丢失。因此，我们不能依赖本地磁盘来写入日志文件。

正如我们在第 1 章“解密微服务”中所讨论的那样，应用的十二因素其中之一是避免把日志文件通过应用自己来路由或者存储。微服务会运行在各个独立的物理或者虚拟机器上，从而导致碎片化的日志文件。在这种情况下，追踪多个微服务间的端到端事务几乎是不可能的。

正如图 7-2 所示，每个微服务生成日志文件到本地文件系统。在本例中，微服务 M1 调用 M3。这些服务把它们的日志写到自己的本地文件系统。这使得关联和理

解这些端到端事务流变得更加困难。而且，有两个 M1 的实例和两个 M3 的实例运行在两个不同的机器上，给服务级别收集日志带来了更多挑战。

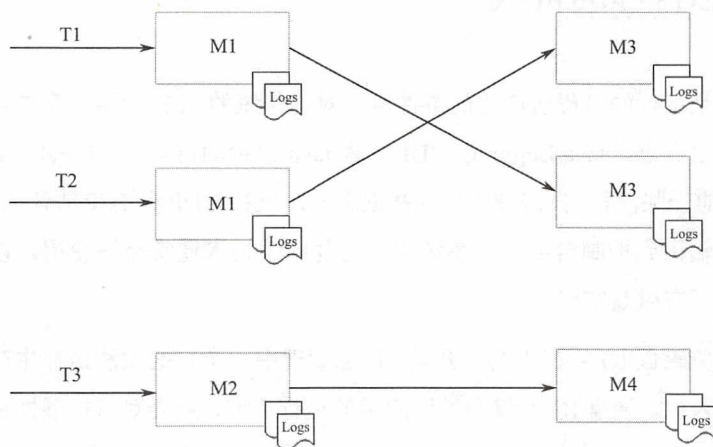


图 7-2

集中式日志解决方案

为了解决前面提到的这些挑战，传统的日志解决方案需要仔细地斟酌。一个新的解决方案，除了解决前面提到的难题，还需要支持下面所总结的这些能力：

- 在日志消息的顶层收集所有日志消息和对其进行分析的能力。
- 关联和追踪端到端事务的能力。
- 更长久地保存日志信息的能力，以用于分析趋势和预测。
- 排除对本地磁盘系统依赖的能力。
- 聚集来自不同源头，如网络设备、操作系统、微服务等日志信息的能力。

这些难题的解决方式是集中存储和分析所有日志消息，不管这些日志来源于哪里。新的解决方式所采取的基本原则是从服务执行环境中把存储和处理日志分离出来。与在微服务的执行环境相比，大数据解决方案更适合于存储和处理数据量大的日志消息。

在集中式日志解决方案中，日志消息会从执行环境中运往一个中央数据仓库，使用大数据解决方案来进行日志的分析和处理。

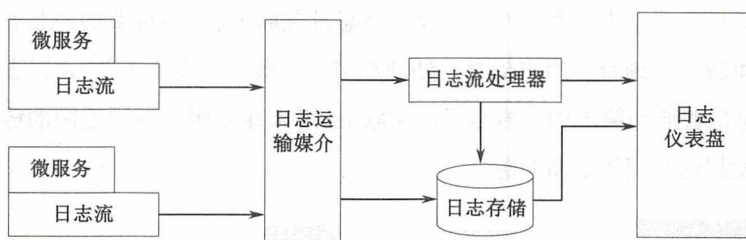


图 7-3

如图 7-3 所示的逻辑图所示，在集中式日志解决方案中有一些组件，例如：

- 日志流：这些是来自源系统的日志消息流。这些源系统包括微服务、其他应用或者网络设备。在典型的基于 Java 的系统中，它们相当于 Log4j 日志消息流。
- 日志运输媒介：日志运输媒介负责收集来自不同源头或者端点的日志消息。日志运输媒介接着把这些消息送往另一些端点，比如写入数据库、推送到仪表盘，或者把它们发送到流处理端点用于后续实时处理。
- 日志存储：日志存储用来将所有日志消息储存起来用于实时分析、预测等。一般来说，一个数据存储是一个 NoSQL 数据库，用来处理大的数据量，比如 HDFS。
- 日志流处理器：日志流处理器能够分析实时日志事件用于做出快速决策。流处理器的行为包括发送信息到仪表盘、发送警告等。在自治的系统中，流处理器甚至能够采取措施来纠正错误。
- 日志仪表盘：仪表盘就是一个显示内容的窗格，用来展示日志分析结果，比如图片和表格。这些仪表盘意味着有关操作和管理的事务。

这种集中管理的优点在于，不存在本地 I/O 或者阻塞性磁盘写入，而且还不使用本地机器的磁盘空间。这种架构基本上类似于用于大数据处理的 lambda 架构。

提示

想要进一步了解 lambda 架构，请访问 <http://lambda-architecture.net>。

在每个日志消息中含有一个上下文、消息及关联 ID 是很重要的。上下文通常拥有一个时间戳、IP 地址、用户信息、处理细节（如服务、类及函数）、日志类型、分类等。消息是普通和简单的文本信息。关联 ID 用来建立服务调用之间的链接，以便能够追踪发起这些调用的微服务。

日志方案的选择

有很多种选择可以用来实现集中式日志解决方案。这些方案使用不同的方法、架构及技术。理解所需要的能力然后根据需求选择正确的方案是至关重要的。

云服务

有很多云日志服务可供使用，如 SaaS 方案。

Loggly 是最受欢迎的基于云的日志服务中的一员。Spring Boot 微服务可以使用 Loggly 的 Log4j 和 Logback 直接把日志消息导入 Loggly 服务。

如果应用或者服务部署在 AWS 上，AWS CloudTrail 可以集成 Loggly，用来进行日志分析。

Papertrail、Logsense、Sumo Logic、Google Cloud Logging 及 Logentries 是另一些基于云日志解决方案的技术。

云日志服务通过给它们提供易于集成的服务，解决了管理复杂的基础设施和大型存储所存在的难题。但是，在选择把云日志作为服务时，延迟是一个不得不考虑的因素之一。

现成的解决方案

已经有很多针对性的工具可以提供端到端日志管理能力，它们在本地安装在一个内部数据中心或者云里。

Graylog 是一个很受欢迎的开源日志管理方案。Graylog 使用 Elasticsearch 来存储日志，MongoDB 作为元数据存储。Graylog 还使用 GELF 库处理 Log4j 日志流。

Splunk 是一款在日志管理和分析领域广受欢迎的商业工具。Splunk 在收集日志时使用日志文件运输的方式，而不是其他解决方案中常用的日志流传输的处理方式。

最佳整合

最后一种方法是挑选最佳的组件来建立一个自定义日志解决方案。

日志传输工具

有很多可以进行日志传输的工具，它们可以跟其他工具一起建立一个端到端的日志管理方案。不同的日志传输工具拥有不同的功能。

Logstash 是一个强大的数据管道工具，可以用来收集和运输日志文件。Logstash 扮演着代理的角色，提供一种接收从不同源头流入的数据，然后把它们同步到不同目的地的机制。Log4j 和 Logback appender 同样可以用来直接从 Spring Boot 微服务发送日志消息到 Logstash。Logstash 的另一端连接到 Elasticsearch、HDFS 或者其他数据库。

Fluentd 是另一个与 Logstash 相当类似的工具，同样的还有 Logspout，但是后者更适合于基于 Docker 容器的环境。

日志流处理器

流处理技术可选择用于即时处理日志流。例如，如果对于某个服务请求，不断的出现 404 错误，这意味服务出了什么问题。这种情况需要尽快进行解决。流处理器很善于处理这种情况，因为它们能够对特定事件流做出反应而传统的反应分析则不能。

流处理中一个典型的架构是结合 Flume 和 Kafka 配合使用 Storm 或者 Spark。Log4j 有 Flume appenders，它在收集日志消息上很有用。这些消息被推到分布式 Kafka 消息队列。流处理器从 Kafka 收集数据然后在把它们送往 Elasticsearch 和其他日志存储之前快速地处理它们。

Spring Cloud Stream、Spring Cloud Stream Modules 及 Spring Cloud Data Flow 同样可以用来创建日志流处理。

日志存储

实时日志消息通常存储在 Elasticsearch 中。Elasticsearch 允许客户端基于文本索引的查询。除了 Elasticsearch 之外，HDFS 同样常用于存储存档日志消息。MongoDB 或者 Cassandra 用来存储总结数据，如月度汇总交易数。离线日志处理可以通过 Hadoop 的 MapReduce 程序来实现。

仪表盘

在中央日志解决方案中所需的最后一步是仪表盘。日志分析最为常用的仪表盘是处于 Elasticsearch 数据存储顶端的 Kibana。Graphite 和 Grafana 同样用来展示日志分析报告。

一个自定义的日志管理实现

前面提到的工具可以用来构建一个自定义的端到端日志解决方案。自定义日志管理中最常使用的架构是配合使用 Logstash、Elasticsearch 和 Kibana。这种架构也就是常说的 ELK 栈。

提示

本章完整的代码可以从代码文件的第 7 章项目下得到。复制 chapter5.configserver、chapter5.eureka-server、chapter5.search、chapter5.search-apigateway 以及 chapter5.website 到一个新的 STS 工作区，然后把它们命名为 chapter7.*。

图 7-4 展示了日志监控流程：

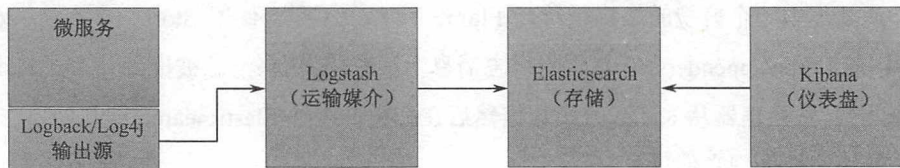


图 7-4

在本节中,我们会尝试使用 ELK 栈来对一个自定义日志解决方案进行简单实现。

遵循这些步骤来实现用于日志的 ELK 栈:

(1) 从 <https://www.elastic.co> 下载并安装 Elasticsearch、Kibana 及 Logstash。

(2) 更新 Search 微服务 (chapter7.search)。审查代码并确保在 Search 微服务中含有一些日志声明。日志声明只是一些使用 slf4j 的简单的日志声明语句:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
//other code goes here
private static final Logger logger = LoggerFactory.
getLogger(SearchRestController.class);
//other code goes here
logger.info("Looking to load flights...");
for (Flight flight : flightRepository.findByOriginAndDestinationAndFlightDate
("NYC", "SFO", "22-JAN-16")) {
    logger.info(flight.toString());
}
```

(3) 在 Search 服务的 pom.xml 文件中添加 logstash 依赖来集成 logback 到 Logstash 中, 如下所示:

```
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>4.6</version>
</dependency>
```

(4) 同时, 降低 logback 的版本来和 Spring 1.3.5.RELEASE 版本兼容:

```
<logback.version>1.1.6</logback.version>
```

(5) 覆盖默认的 Logback 配置。可以通过在 src/main/resources 目录下增加一个新的 logback.xml 文件来实现:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<include resource="org/springframework/boot/logging/logback/defaults.xml"/>
<include resource="org/springframework/boot/logging/logback/console-appender.xml" />
<appender name="stash" class="net.logstash.logback.appender.LogstashTcpSocketAppender">
```



```
<destination>localhost:4560</destination>
<!-- encoder is required -->
<encoder class="net.logstash.logback.encoder.LogstashEncoder" />
</appender>
<root level="INFO">
  <appender-ref ref="CONSOLE" />
  <appender-ref ref="stash" />
</root>
</configuration>
```

前面的配置通过增加一个新的 TCP socketappender 来覆盖默认的 Logback 配置，它把所有的日志消息导向了一个 Logstash 服务，这个服务监听 4560 端口。正如前面配置中所提到的那样，还需要添加一个编码器。

(6) 像下面的代码所展示的那样创建一条配置，然后把它保存在 Logstash.conf 文件中。这个文件的地址无关紧要，因为它会在启动 Logstash 之后作为一个参数进行传递。这条配置会从监听 4560 端口的 socket 中得到输入，然后把输出发送到监听 9200 端口的 Elasticsearch。stdout 是可选的，并且它被设置为 debug 模式：

```
input {
  tcp {
    port => 4560
    host => localhost
  }
}
output {
  elasticsearch { hosts => ["localhost:9200"] }
  stdout { codec => rubydebug }
}
```

(7) 分别从它们各自的安装目录中运行 Logstash、Elasticsearch 及 Kibana：

```
./bin/logstash -f logstash.conf
./bin/elasticsearch
./bin/kibana
```

(8) 运行 Search 微服务。它会调用单元测试然后会把前面提到的日志声明打印出来。

(9) 打开浏览器，网址输入 <http://localhost:5601> 访问 Kibana。

(10) 单击 Settings | Configure an index pattern, 如图 7-5 所示。

Configure an index pattern

In order to use Kibana you must configure at least one index pattern. Index patterns are used to identify the Elasticsearch index to run search and analytics against. They are also used to configure fields.

☒ Index contains time-based events
☐ Use event times to create index names

Index name or pattern
 Patterns allow you to define dynamic index names using * as a wildcard. Example: logstash-*

logstash-*

Time-field name
 @timestamp

图 7-5

(11) 单击 Discover 菜单来查看日志。如果一切顺利,我们会看到如下面所示的 Kibana 截图。需要注意的是,日志消息是展示在 Kibana 屏幕上。Kibana 提供开箱即用的特点来使用日志消息创建概要图表,如图 7-6 所示。

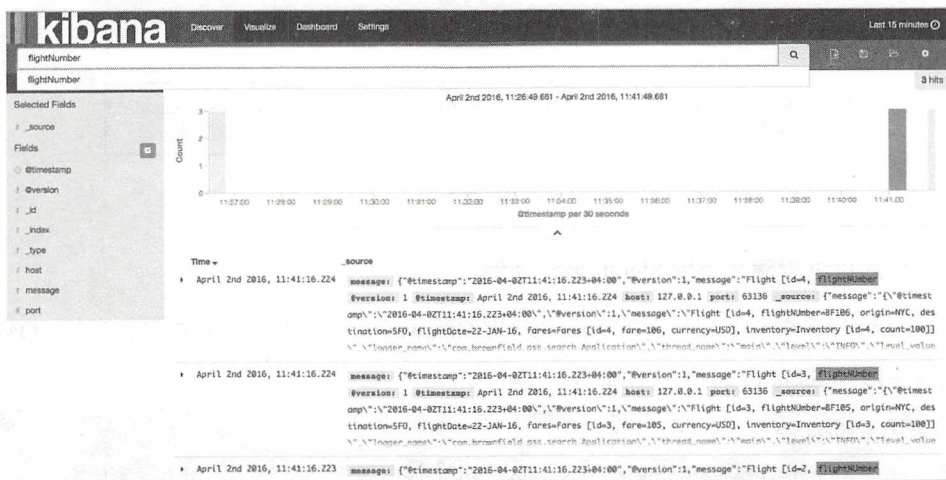


图 7-6

使用 Spring Cloud Sleuth 进行分布式跟踪

上一节通过集中日志数据,解决了微服务分布式和碎片化的日志问题。有了中

央日志解决方案，我们可以在中央存储器存入所有日志。然而追踪端到端的事务仍然几乎是不可能的。为了做到端到端追踪，事务生成微服务需要有对应的 ID。

Twitter 的 Zipkin, Cloudera 的 Htrace 及 Google 的 Dapper 系统就是分布式追踪系统的例子。Spring Cloud 在这些使用 Spring Cloud Sleuth 库的服务之上提供了一个包装组件。

分布式追踪伴随着 span 和 trace 的概念。span 是工作的一个单元。例如，调用一个由 64 位 ID 标示的服务，一组 span 形成一个树形结构，称为一个 trace。使用 trace ID，这次调用可被端到端追踪：

如图 7-7 所示，Microservice1 调用 Microservice2，Microservice2 调用 Microservice3。在这种情况下，相同的 trace ID 贯穿所有的微服务，可用于端到端事务。为了证明这一点，我们将使用 Search API Gateway 和 Search 微服务。一个新的端点必须被添加到 Search API Gateway (chapter7.search-apigateway)，以便于在内部调用 Search 服务返回数据。没有 trace ID，几乎不可能追踪或联系从 Website 到 Search API Gateway 再到 Search 微服务的调用。在这种情况下，它只涉及两三个服务，而在复杂的环境中，则可能存在许多相互依赖的服务。

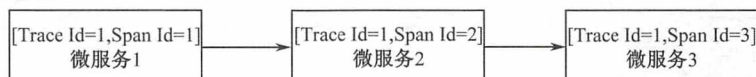


图 7-7

按照以下步骤来使用 Sleuth 创建样例：

(1) 更新 Search 微服务和 Search API Gateway。在此之前，需要在各自的 POM 文件中添加 Sleuth 依赖，参照以下代码：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

(2) 在创建一个新服务时，选择 Sleuth 和 Web，如图 7-8 所示。

(3) 向 Search 服务中添加 Logstash 依赖，同时还需要配置 Logback，就像上个例子中所描述的那样。

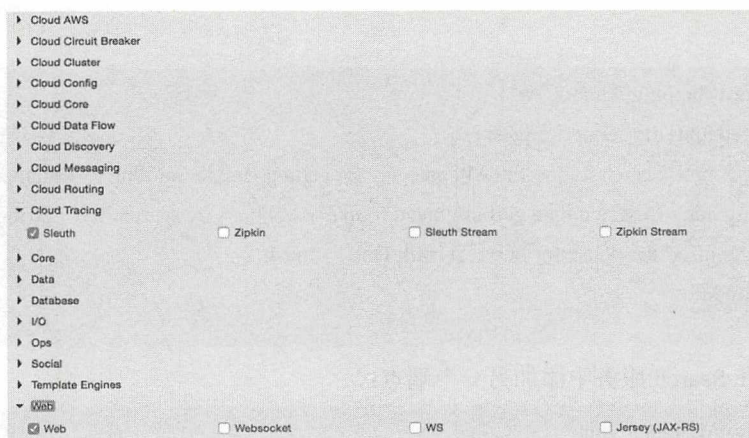


图 7-8

(4) 接下来的步骤是在 Logback 配置中添加两个新的属性：

```
<property name="spring.application.name" value="search-service"/>
<property name="CONSOLE_LOG_PATTERN" value="%d{yyyy-MM-ddHH:mm:ss.SSS}
[${spring.application.name}]
[trace=%X{X-Trace-Id:-},span=%X{X-Span-Id:-}][%15.15t] %-40.40logger{39}: %m%n"/>
```

第一个属性是应用的名称。在这里我们把服务 ID 作为名称，Search 和 Search API Gateway 各自的 ID 分别为 search-service 和 search-apigateway。第二个属性是一个可选的模式，可以用来打印带有一个 trace ID 和 span ID 的控制台日志消息。前面的这项更改需要同时应用在两个服务上。

(5) 添加下面的代码块来建议 Sleuth 在 Spring Boot Application 类中何时开始一个新的 span ID。在本例中，AlwaysSampler 用来表示每当一个请求到达服务时都需要创建一个新的长度 ID：

```
@Bean
public AlwaysSampler defaultSampler() {
    return new AlwaysSampler();
}
```

(6) 给 Search API 网关添加一个新的端点，它会调用 Search 服务。这是为了展示 trace ID 在多个微服务之间的传输。网关中的这个函数通过调用 Search 服务返回了 Hub：

```
@RequestMapping("/hubongw")
String getHub(HttpServletRequest req){
    logger.info("Search Request in API gateway for getting Hub,forwarding to search-service ");
    String hub = restTemplate.getForObject("http://search-service/search/hub", String.class);
    logger.info("Response for hub received, Hub "+ hub);
    return hub;
}
```

(7) 在 Search 服务中添加另一个端点:

```
@RequestMapping("/hub")
String getHub(){
    logger.info("Searching for Hub, received from searchapigateway");
    return "SFO";
}
```

(8) 一旦添加进去, 返回两个服务。使用浏览器 (<http://localhost:8095/hubongw>) 在网关 (/hubongw) 端点访问新的 Hub。

前面曾经提到, Search API Gateway 服务运行在 8095 端口, Search 服务运行在 8090 端口。

(9) 通过控制台日志来查看打印的 trace ID 和 span ID。第一个输出来自 Search API Gateway, 第二个来自 Search 服务。需要注意的是, trace ID 在两种情况下是相同的:

```
2016-04-02 17:24:37.624 [search-apigateway][trace=8a7e278f-7b2b-43e3-a45c-69d3ca66d663,
span=8a7e278f-7b2b-43e3-a45c-69d3ca66d663][io-8095-exec-10]
c.b.p.s.a.SearchAPIGatewayController :Response for hub received, Hub DXB
2016-04-02 17:24:37.612 [search-service][trace=8a7e278f-7b2b-43e3-a45c-69d3ca66d663,span=
fd309bba-5b4d-447f-a5e1-7faab90cfb1][nio-8090-exec-1]
c.b.p.search.component.SearchComponent :Searching for Hub, received from search-apigateway
```

(10) 打开 Kibana 控制台, 然后使用在控制台上打印的 trace ID 搜索 trace ID。在本例中, trace ID 是 8a7e278f-7b2b-43e3-a45c-69d3ca66d663。如图 7-9 所示, 有了 trace ID, 我们就可以在多个服务之间追踪某个服务。

典型问题总结如下：

- 统计信息和度量标准在许多服务、实例和机器上是分段的。
- 微服务可能使用多样化的技术来实现，这使得事情变得更复杂。单个监控工具可能不能提供所有所需的监控项。
- 微服务部署拓扑是动态的，使之不可能提前配置服务器、实例和监控参数。

许多传统监控工具能很好地监控单体应用，但是做不到监控大规模的、分布式的、内部相互紧密关联的微服务系统。许多传统监控系统是基于代理的，即在目标机器或者应用实例上提前安装代理。这也造成两个挑战：

- 如果代理需要与服务或操作系统深度集成，这将难以管理一个动态环境。
- 如果这些工具在监控或检测应用时增加系统开销，将会导致性能问题。

许多传统工具需要基线度量指标。这样的系统依据提前设置的规则来工作，例如，如果 CPU 利用率超过 60% 并且保持这一水平有 2min，应该发给管理员一个警告。在大型可网络扩展的部署下是很难提前配置这些值的。

新一代监控应用通过他们自己学习应用行为并自动设置阈值。这使得管理员从这些普通任务中释放出来。自动基线标准有时比人工预测更精确。

如图 7-10 所示，微服务监控的关键领域如下：

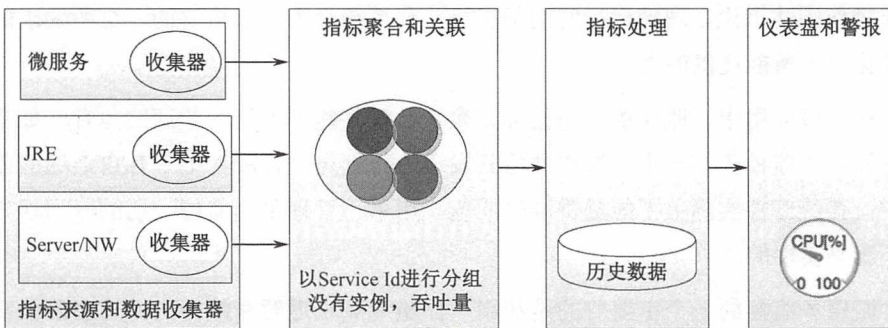


图 7-10

- 指标来源和数据收集：在源端的指标收集通过服务器推送指标信息到一个集中收集器中或通过嵌入的轻量级代理来收集信息。数据收集器从不同的源收集监控指标，如网络、物理机器、容器、软件组件、应用等。面临的

挑战就是使用自动覆盖机制替代静态配置来收集这些数据。

这些通过在源机器上运行代理并从源上流收集数据或者定期轮询来实现。

- 度量指标聚合和关联：聚合能力是必须的，用于聚合从不同源收集来的指标，如用户交易、服务、基础设置、网络等。聚合也是有挑战的，由于它需要在某种程度上理解应用的行为，如服务依赖、服务分组等。在许多情况下，这些是基于源提供的元数据自动生成的。

通常，这是由接受这些指标的中间层来完成的。

- 处理指标和可操作的见解：一旦数据聚合了，下一步要做的是度量。度量的典型做法是设置阈值。在新一代监控系统，这些阈值是自动被发现的。监控工具分析数据并提供可操作的见解。

这些工具可能使用大数据和流式分析解决方案。

- 警报、操作和仪表板：一旦检测到问题，必须通知相关人员或系统。不像传统系统，微服务监控系统应该具有实时采取行动的能力，主动的监控是实现自我修复的本质，仪表板用于展示 SLAs 和 KPIs 等。

仪表板和警报工具能够处理这些要求。

微服务监控典型地通过 3 个方法完成。为确保监控有效，这些组合是不可或缺的。

- 应用性能监控（APM）：这更多地是系统度量收集、处理、警报和仪表板呈现的传统方法，这些更多是来自系统的角度。应用拓扑发现和可视化是许多 APM 工具实现的新功能，不同 APM 提供商之间的能力不同。
- 综合监控：这是一种用于在生产或类似生产环境中使用具有多个测试场景的端到端事务来监视系统行为的技术。收集数据以验证系统的行为和潜在热点。合成监视有助于了解系统依赖性。
- 真实用户监控（RUM）或用户体验监控：这通常是一个基于浏览器的软件，用于记录真实的用户统计信息，如响应时间、可用性和服务级别。对于微服务，具有更频繁的发布周期和动态拓扑，用户体验监控更为重要。

监控工具

有许多工具可用于监控微服务，这些工具之间很多有重叠部分。监控工具的选择实际依赖于需要被监控的生态系统。在大多数案例中，整个微服务系统需要不止一个监控工具。

这一节的目标是通过一些常用的对微服务友好的监控工具使我们更熟悉微服务的监控：

- AppDynamics、Dynatrace 和 New Relic 是在 APM 领域的顶级商业供应商，根据于 Gartner Magic Quadrant 2015。这些工具是微服务友好并且高效支持单个控制台的微服务监控。Ruxit、Datadog 和 Dataloop 是为分布式系统专门构建的其他商业产品，基本上是微服务友好的。多个监视工具可以使用插件向 Datadog 提供数据。
- 云供应商提供自己的监控工具，但在许多情况下，这些监控工具可能不足以进行大规模微服务监控。例如，AWS 使用 CloudWatch，Google Cloud Platform 使用云监控从各种来源收集信息。
- 一些数据收集库（如 Zabbix、statd、collectd、jmxtrans 等）在收集运行时间统计信息、度量标准、计量器和计数器时运行在较低级别。通常此信息被馈送到数据收集器和处理器，如 Riemann、Datadog 和 Librato，或仪表板，如 Graphite。
- Spring Boot Actuator 是收集微服务指标、计量器和计数器的好工具之一，我们在第 2 章“用 Spring Boot 构建微服务”中讨论过。Netflix Servo 是类似于 Actuator 的度量收集器，QBit 和 Dropwizard 度量也属于同一类别的度量收集器。所有这些度量收集器都需要一个聚合器和仪表板来促进全面监控。
- 通过日志记录进行监视是比较流行的，但在微服务监视中是一种不太有效的方法。在这种方法中，如上一节所讨论的，日志消息从各种来源（如微服务、容器、网络等）发送到中央位置。然后我们可以使用日志文件跟踪事务、识别热点等。Loggly、ELK、Splunk 和 Trace 是此方面中的候选项。
- Sensu 是开源社区微服务监控的流行选择。Weave Scope 是另一种工具，主要针对容器化部署。Spigo 是与 Netflix 堆栈紧密匹配的专用微服务监控系统之一。

- Pingdom、New Relic Synthetics、Runscope、Catchpoint 等为实时系统上的综合事务监视和用户体验监视提供选项。
- Circonus 更多地被分类为 DevOps 监控工具，但也可以进行微服务监控。Nagios 是一个流行的开源监控工具，但更多地落入传统的监控系统。
- Prometheus 提供了一个时间序列数据库和可视化 GUI，用于构建自定义监视工具。

监控微服务依赖

当有大量具有依赖关系的微服务时，有一个监视工具可以显示微服务之间的依赖关系很重要。静态配置和管理这些依赖关系不是一种可扩展的方法。有许多工具可用于监视微服务依赖关系，如下所示：

- 辅助工具（如 AppDynamics、Dynatrace 和 New Relic）可以在微服务之间绘制依赖关系。端到端事务监视还可以跟踪事务依赖关系。其他监视工具（如 Spigo）也可用于微服务依赖关系管理。
- CMDB 工具（如 Device42）或专用工具（如 Accordance）可用于管理微服务的依赖关系。Veritas Resiliency Platform (VRA) 也可用于基础设施发现。
- 使用 Graph 数据库（如 Neo4j）的自定义实现也很有用。在这种情况下，微服务必须预配置其直接和间接依赖性。在服务启动时，它使用 Neo4j 数据库发布和交叉检查其依赖性。

Spring Cloud Hystrix 用于容错微服务

本节将探讨 Spring Cloud Hystrix 作为容错和延迟容忍微服务实现的库。Hystrix 基于快速故障和快速恢复原则。如果服务有问题，Hystrix 有助于隔离它。它有助于通过回退到另一个预配置的后备服务来快速恢复。Hystrix 是 Netflix 的另一个经过测试的库。Hystrix 基于熔断器模式。

提示

有关熔断器模式的更多信息，请访问 <https://msdn.microsoft.com/enus/library/dn589784.aspx>。

在本节中，我们将搭建一个带有 Spring Cloud Hystrix 的熔断器。执行以下步骤

更改 Search API Gateway 服务以将其与 Hystrix 集成:

(1) 更新 Search API Gateway 服务。添加 Hystrix 依赖到服务中。如果是从 scratch 开发, 选择如图 7-11 所示。

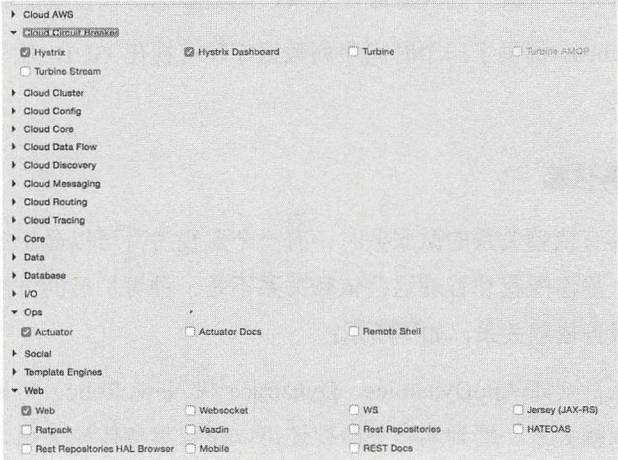


图 7-11

(2) 在 Spring Boot Application 类中添加@EnableCircuitBreaker 注解。这个指令会告诉 Spring Cloud Hystrix 为这个应用开启一个熔断器。它还暴露了度量收集的 /hystrix.stream 端点。

(3) 使用一个方法向 Search API Gateway 服务添加组件类; 在这个例子中是用 @HystrixCommand 注解的 getHub 方法。这告诉 Spring 这个方法容易失败。Spring Cloud 库包装这些方法通过开启熔断器去处理容错和延时。Hystrix 命令一般跟随着一个 fallback 方法。如果发生故障, Hystrix 会自动启用所提到的回退方法, 并将流量转移到回退方法。如下面代码所示, 在这个例子中, getHub 会回退到 getDefaultHub:

```
@Component
class SearchAPIGatewayComponent {
    @LoadBalanced
    @Autowired
    RestTemplate restTemplate;
    @HystrixCommand(fallbackMethod = "getDefaultHub")
    public String getHub() {
        String hub = restTemplate.getForObject("http://search-service/search/hub", String.class);
```

```

        return hub;
    }

    public String getDefaultHub() {
        return "Possibly SFO";
    }
}

```

(4) SearchAPIGatewayController 的 getHub 方法调用 SearchAPIGatewayComponent 的 getHub 方法, 如下:

```

@RequestMapping("/hubongw")
String getHub(){
    logger.info("Search Request in API gateway for getting Hub, forwarding to search-
service");
    return component.getHub();
}

```

(5) 本练习的最后一部分是构建 Hystrix 仪表板。构建这个应用程序时包括 Hystrix、Hystrix Dashboard 和 Actuator。

(6) 在 Spring Boot Application 类, 添加@EnableHystrixDashboard 注解。

(7) 启动 Search 服务, 搜索 API Gateway 和 Hystrix Dashboard 应用。在浏览器上打开指向 Hystrix Dashboard 应用程序的 URL。在这个例子中, Hystrix Dashboard 在 9999 端口启动。访问 URL: <http://localhost:9999/hystrix>。

(8) 将显示类似于以下截图的屏幕。在 Hystrix 仪表板中, 输入要监控服务的 URL。在这个用例中, Search API Gateway 运行在 8095 端口。因此, hystrix.stream URL 是 <http://localhost:8095/hystrix.stream>, 如图 7-12 所示。

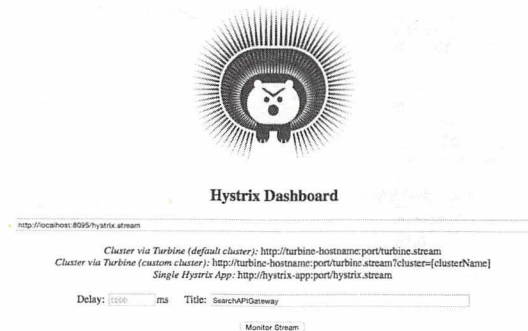


图 7-12

(9) Hystrix 仪表板会显示如图 7-13 所示。

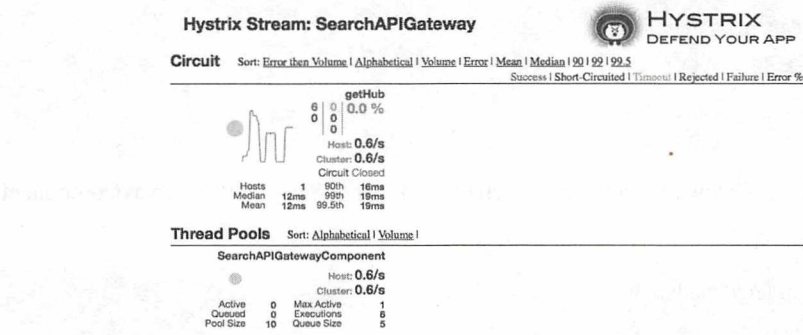


图 7-13

提示

注意，必须执行至少一个事务才能看到显示。这可以通过单击 <http://localhost:8095/hubongw> 来完成。

(10) 通过停掉 Search 服务来创建一个故障场景。注意：fallback 方法会在单击 URL <http://localhost:8095/hubongw> 后调用。

(11) 如果存在连续故障，则电路状态将更改为打开。这可以通过多次击打前面的 URL 来完成。在打开状态下，将不再检查原始服务。Hystrix 仪表板将显示电路的状态为 Open，如图 7-14 所示。一旦电路被周期性地打开，系统将检查原始服务状态以进行恢复。当原始服务恢复时，断路器将回到原始服务，并且状态将被设置为关闭。

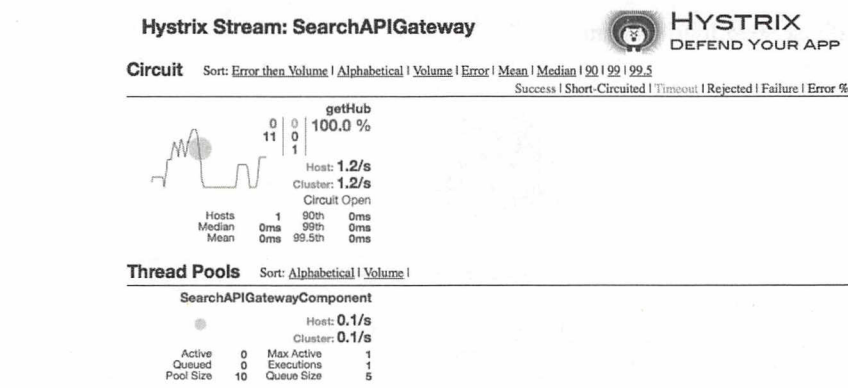


图 7-14

提示

要了解这些参数的含义，请访问 Hystrix wiki，网址为 <https://github.com/Netflix/Hystrix/wiki/Dashboard>。

用 Turbine 聚集 Hystrix 流

在上一个示例中，我们的微服务的/hystrix.stream 端点在 Hystrix 仪表板中给出。Hystrix 仪表板一次只能监视一个微服务。如果有许多微服务，那么指向该服务的 Hystrix 仪表板必须在每次我们切换微服务进入监控时更改。一次查看一个实例是比较乏味的，特别是当有一个微服务或多个微服务的许多实例时。

我们必须有一个机制来聚合来自多个/hystrix.stream 实例的数据，并将其合并到一个仪表板视图中。Turbine 正是做这样的事情。Turbine 是另一个从多个实例收集 Hystrix 流并将它们合并到一个/turbine.stream 实例中的服务器。现在，Hystrix 仪表板可以指向/turbine.stream 以获取合并信息，如图 7-15 所示。

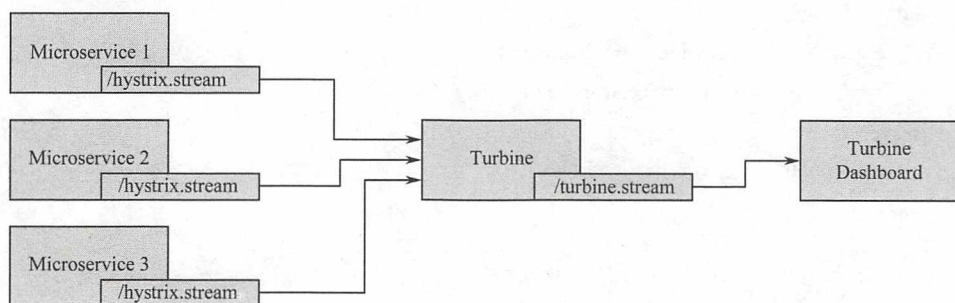


图 7-15

提示

Turbine 当前只能根据不同主机名来工作。每个实例必须运行在独立的主机上。如果您在本地一个主机上测试多服务，更新主机文件 (/etc/hosts) 成类似多主机。一旦完成，bootstrap.properties 配置如下：

```
eureka.instance.hostname: localadmin2
```

此示例展示如何使用 Turbine 监视跨多个实例和服务的断路器。在此示例中，我们将使用 Search 服务和 Search API Gateway 来演示。

执行以下步骤来构建和执行此示例：

(1) 可以使用 Spring Boot Starter 将 Turbine 服务器创建为另一个 Spring Boot 应用程序。选择 Turbine 以包括 Turbine 库。

(2) 创建应用程序后，将@EnableTurbine 添加到主 Spring Boot 应用程序类。在此示例中，Turbine 和 Hystrix Dashboard 都配置为在同一个 Spring Boot 应用程序上运行。这可以通过向新创建的 Turbine 应用程序添加以下注解来实现：

```
@EnableTurbine
@EnableHystrixDashboard
@SpringBootApplication
public class TurbineServerApplication {
```

(3) 将以下配置添加到.yaml 或属性文件中，以指向我们感兴趣监视的实例：

```
spring:
  application:
    name : turbineserver
  turbine:
    clusterNameExpression: new String('default')
    appConfig : search-service,search-apigateway
  server:
    port: 9090
  eureka:
    client:
      serviceUrl:
        defaultZone: http://localhost:8761/eureka/
```

(4) 上述配置指示 Turbine 服务器查找 Eureka 服务器来获取 search-service 和 search-apigateway 服务。search-service 和 search-apigateway 服务 ID 用于向 Eureka 注册服务。Turbine 使用这些名称通过与 Eureka 服务器检查来解析实际的服务主机和端口。然后它将使用此信息从每个实例读取/hystrix.stream。然后，Turbine 将读取所有单独的 Hystrix 流，聚合所有这些流，并将它们暴露在 Turbine 服务器的/turbine.stream URL 下。

(5) 集群表示指向默认集群，因为在此示例中没有进行明确的集群配置。如果手动配置集群，则必须使用以下配置：


```
turbine:
  aggregator:
    clusterConfig: [comma separated clusterNames]
```

(6) 更改搜索服务的 SearchComponent 以添加另一个熔断器，如下所示：

```
@HystrixCommand(fallbackMethod = "searchFallback")
public List<Flight> search(SearchQuery query){
```

(7) 另外，在搜索服务中的 Application main 类中添加@EnableCircuitBreaker。

(8) 将以下配置添加到 Search 服务的 bootstrap.properties。这是必需的，因为所有服务都在同一主机上运行：

```
Eureka.instance.hostname: localdomain1
```

(9) 同样，在 Search API Gateway 服务的 bootstrap.properties 中添加以下内容。这是为了确保两个服务使用不同的主机名：

```
eureka.instance.hostname: localdomain2
```

(10) 在这个例子中，我们将运行两个 search-apigateway 实例：一个在 localdomain1: 8095，另一个在 localdomain2:8096。我们还将 localdomain1 上运行一个搜索服务实例：8090。

(11) 使用命令行覆盖运行微服务以管理不同的主机地址，如下所示：

```
java -jar -Dserver.port=8096 -Deureka.instance.hostname=localdomain2 -Dserver.address=
localdomain2 target/chapter7.searchapigateway-1.0.jar
java -jar -Dserver.port=8095 -Deureka.instance.hostname=localdomain1 -Dserver.address=
localdomain1 target/chapter7.searchapigateway-1.0.jar
java -jar -Dserver.port=8090 -Deureka.instance.hostname=localdomain1 -Dserver.address=
localdomain1 target/chapter7.search-1.0.jar
```

(12) 在浏览器中访问<http://localhost:9090/hystrix>打开 Hystrix Dashboard。

(13) 对比之前的/hystrix.stream，这一次，我们将指向/turbine.stream。在这个例子中，Turbine 流在 9090 上运行。因此，在 Hystrix 仪表板中给出的 URL 是<http://localhost:9090/turbine.stream>。

(14) 打开：<http://localhost:8095/hubongw>和<http://localhost:8096/hubongw>。仪表板页面将显示 getHub 服务。

(15) 运行 chapter7.website。访问http://localhost:8001去执行搜索事务。执行完后，仪表板页面也将显示搜索服务，如图 7-16 所示。

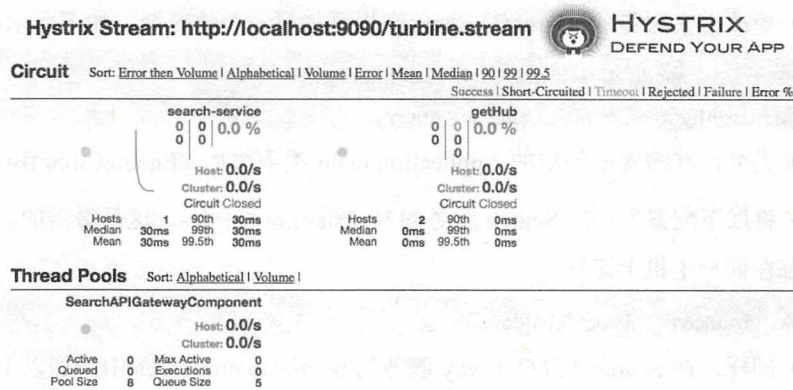


图 7-16

我们可以在仪表板中看到，搜索服务来自 Search 微服务，getHub 来自 Search API 网关。由于我们有两个 Search API 网关实例，getHub 来自两个主机，如上图中 Hosts 2 所示。

使用数据湖泊的数据分析

与分段日志和监视的情况类似，碎片数据是微服务架构中的另一个挑战。分段数据在数据分析中提出了挑战。此数据可用于简单的业务事件监视、数据审计，甚至从数据中挖掘出商业智能。

数据湖泊或数据中心是处理这种情况的理想解决方案。事件化架构模式通常用于分享状态，当状态更改时，微服务将状态更改作为事件发布到可订阅的中间件中。有意者可以订阅这些活动并根据他们的要求进行处理。中央事件存储器还可以预订这些事件，并将它们存储在大数据系统中以用于进一步分析。

图 7-17 中显示了这种数据处理的常用架构之一。

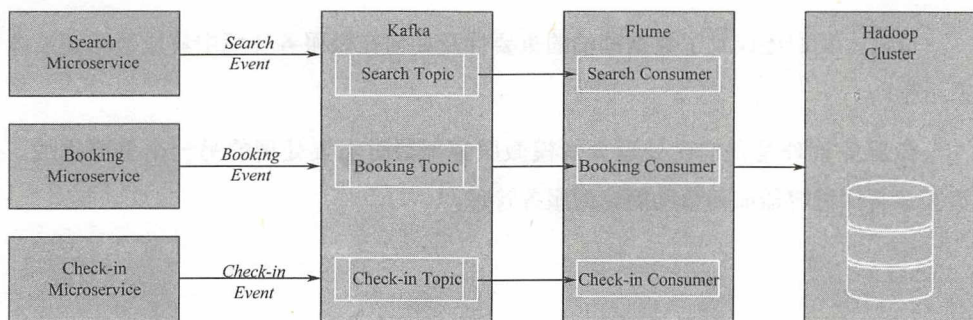


图 7-17

从微服务生成的状态更改事件（在我们的例子中，Search、Booking 和 Check-In 事件）被推送到分布式高性能消息传递系统（如 Kafka）。数据摄取服务（如 Flume）可以订阅这些事件并将其更新到 HDFS 集群。在某些情况下，这些消息将由 Spark Streaming 进行实时处理。为了处理异构事件源，Flume 也可以在事件源和 Kafka 之间使用。

Spring Cloud Streams、Spring Cloud Streams 模块和 Spring Data Flow 也可用作高速数据摄取的替代方法。

总结

在本章中，您了解了处理互联网级微服务时日志记录和监视方面的挑战。

我们探讨了集中日志的各种解决方案。您还了解了如何使用 Elasticsearch、Logstash 和 Kibana（ELK）实现自定义集中式日志记录。为了理解分布式跟踪，我们使用 Spring Cloud Sleuth 升级了 BrownField 微服务。

在本章的第二部分，我们深入探讨了微服务监控解决方案和不同监控方法所需的功能。随后，我们审查了一些可用于微服务监控的工具。

BrownField 微服务被 Spring Cloud Hystrix 和 Turbine 进一步增强，以监视服务间通信的延迟和故障。这些示例还演示了如何在故障情况下使用断路器模式回退到另一个服务。

最后，我们还谈到了数据湖泊的重要性及如何在微服务环境中集成数据湖泊体系结构。

微服务管理是我们在处理大规模微服务部署时需要处理的另一个重要挑战。第 8 章将探讨容器如何帮助简化微服务管理。

第 8 章

用 Docker 实现容器化微服务



在微服务的领域中，容器化部署简直是锦上添花。它帮助微服务更加独立自主地运行在基础设施里面，从而使得微服务能够运行在云平台上。

本章将介绍容器和相关虚拟机镜像的概念，以及微服务的容器化部署。然后将进一步让读者熟悉如何建立 Docker 镜像，以及如何用 Spring Boot 和 Spring Cloud 去开发 BrownField PSS 微服务。

最后，本章也将涉及如何管理、维护和部署 Docker 镜像到生产环境。

本章末尾，您将学到：

- 容器化的概念及其在微服务环境中的相关性。
- 构建和部署微服务作为 Docker 镜像和容器存储。
- 使用 AWS 作为基于云平台的 Docker 部署的示例。

回顾微服务功能模型

在本章中，我们将从第 3 章“微服务概念的应用”中讨论的微服务能力模型探索以下微服务功能：

- 容器与虚拟机。
- 私有云和公有云。
- 微服务镜像库。

模型如图 8-1 所示。

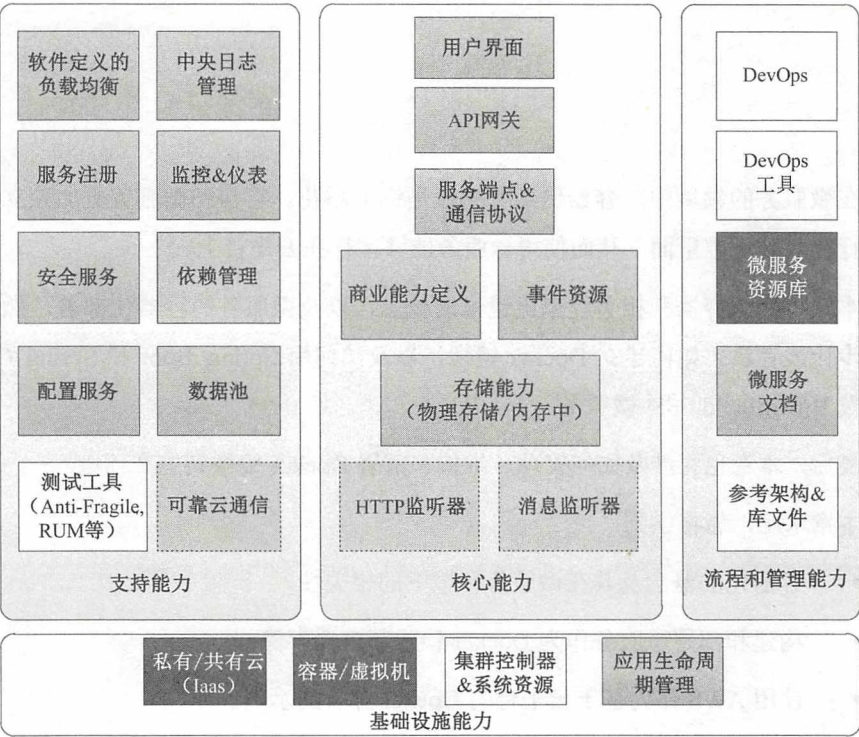


图 8-1

理解 BrownField PSS 微服务的区别

在第 5 章“通过 Spring Cloud 对微服务进行扩（缩）容”中，BrownField PSS 微服务是使用 Spring Boot 和 Spring Cloud 开发的。这些微服务打成本地开发机器上运行的 jar 包进行部署。

在第 6 章“自动化扩（缩）容微服务”中，在自定义生命周期管理器里面添加了自动扩展功能。在第 7 章“日志记录和监控微服务”中，使用集中的日志和监控方案来解决记录和监控的问题。

在我们的 BrownField PSS 实现中仍然有一些缺陷。到目前为止，微服务运行在没有使用任何云基础设施之上。独立的机器，传统的应用部署方式，并不是微服务部署的最好解决方案。自动化如自动配置、按需伸缩的负载均衡能力、自助服务和基于已使用的资源来支付的模式是高效管理大规模微服务部署所必备的功能。一般来说，云基础架构已提供了所有微服务所需的基本功能。因此，具有提前预知能力的私有云或公共云更适合部署互联网规模的微服务。

此外，对每个裸机运行一个微服务实例不是一个有效控制成本的方法。因此，在大多数情况下，企业会在单个裸机服务器部署多个微服务。在单个裸机上运行多个微服务可能会有一些“嘈杂的邻居”的问题。在同一台机器上运行的微服务实例之间没有任何隔离。因此，部署了在单个机器上的服务可能会占用其他服务的空间，从而影响微服务各自运行的性能。

另一种方法是在 VM 上运行微服务。但是，VM 自身太重了。因此在物理机上运行许多较小的 VM 不是资源节省最有效的方法，反而导致资源浪费。如果是共享一个虚拟机来部署多个服务，我们最终会面临与共享裸机同样的问题。

在基于 Java 的微服务的情况下，共享 VM 或裸机部署多个微服务也会导致微服务之间共享一个 JRE。这是因为在我们的 BrownField PSS 中创建的是 jar 包，只抽象应用程序级别的代码和依赖，而不是一整个 JRE。机器上安装的 JRE 一旦有任何的更新，将对部署在此机器上的所有微服务产生影响。类似地，如果有操作系统级别

的参数、库或特定微服务所需的调整，很难在共享 JRE 的环境中有效地管理它们。

微服务的原则应该是自主封装在其端到端的运行环境中，自给自足并且没有任何依赖。基于这个原则，所有组件，如 OS、JRE 和微服务二进制文件，必须是自给自足和独立的。实现这一点的唯一选择是遵循每个 VM 部署一个微服务的方法。但是，这将导致 VM 资源未充分利用，并且在许多情况下，由于这种额外的开销，反而抵消了微服务自身拥有的益处。

什么是容器

容器不是革命性的、开拓性的概念。它已经存在相当一段时间了。由于云计算的广泛采用，容器重新进入了人们的视野。传统虚拟机在云计算空间中存在的缺点也加速了容器技术的推广。例如，Docker 的容器提供者在很大程度上简化了容器技术，使得容器越来越流行。最近 DevOps 和微服务的流行也促进了容器技术重生。

那么，什么是容器？容器在操作系统之上提供私有空间。这种技术也称为操作系统虚拟化。容器的本质就是在操作系统的内核提供隔离的虚拟空间。每个这样的虚拟空间称为容器或虚拟引擎（VE）。容器允许进程在主机操作系统之上的隔离环境中运行。在同一主机上运行的多个容器如图 8-2 所示。

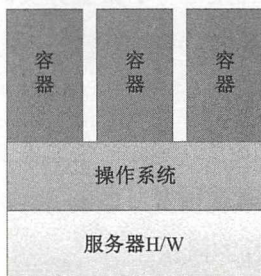


图 8-2

容器是构建、传输和运行分区软件组件的简单机制。通常容器打包所有对于运行应用程序至关重要的二进制文件和库。容器保留自己的文件系统、IP 地址、网络接口、内部进程、命名空间、操作系统库、应用程序二进制文件、依赖关系和其他应用程序配置。

各个组织使用了数十亿个容器。此外，许多大型组织对容器化技术投入大量精力。由于许多大型操作系统供应商和云提供商的支持，Docker 容器已经处于远远领先的地位。例如，也有一些其他的容器化解决方案，包括 Lmctfy、SystemdNspawn、Rocket、Draw 桥接、LXD、Kurma 和 Calico。目前关于容器开源的规范也在制定中。

VMs 与容器之间的区别

以前数据中心虚拟化组件通常使用的是 Hyper-V、VMWare 及 Zen 等非常流行的 VMs。企业通过传统裸机上实施虚拟化，节省了大量成本。它还帮助许多企业以更好地利用其现有的基础设施。由于 VMs 支持自动化，许多企业都可以在虚拟机管理上花费更少的精力。虚拟机还帮助应用程序能够在隔离的环境中运行。

到现在，您可能会认为虚拟化和容器化具有完全相同的特性。但是事实并非如此。在 VMs 和容器之间进行同一基准的比较是不公平的。虚拟机和容器解决了不同的虚拟化问题，从图 8-3 中可以看出：

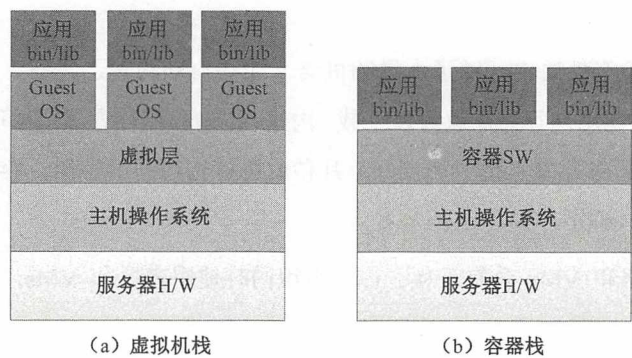


图 8-3

与容器相比，虚拟机更为底层。VMs 提供硬件虚拟化，如 CPU、主板、内存等。VM 是具有嵌入式操作系统（通常称为客户操作系统）的隔离单元。VMs 复制整个操作系统并在 VM 中运行，而不依赖于主机操作系统环境。由于 VMs 嵌入了完整的操作系统环境，以上功能都是重量级的。这其实是一把双刃剑，它的优点就是 VMs 上运行的进程的完全隔离，缺点就是由于 VMs 的资源需求，它限制了可以在裸机中运行的 VMs 的数量。

VM 的大小对启动和停止 VM 的时间有直接影响。由于启动 VM 会同时启动操作系统，VMs 的启动时间通常很长。VMs 对基础架构团队更加友好，因为管理 VMs

就是需要底层的基础架构的能力。

在容器世界中，容器不能模拟整个硬件或操作系统。与虚拟机不同，容器共享主机内核和操作系统的某些部分。容器中没有客户机操作系统的概念。容器直接在主机操作系统之上提供一个隔离的运行环境。这是它的优点也是劣势。优点是它更轻、更快。由于同一台机器上的容器共享主机操作系统，容器的总体资源利用率相当小。最直接的好处就是，相比重量级 VMs，可以在一台机器上运行许多较小的容器。由于同一主机上的容器共享主机操作系统，因此也存在一些限制。例如，不能在容器中设置 iptables 防火墙规则，容器中的进程与运行在同一主机上不同容器上的进程完全独立。

与 VMs 不同，容器镜像在社区门户是开源的。开发人员不必从头开始构建镜像。相反，他们可以从经认证的源获取基本镜像，并在下载的基本镜像之上添加额外的软件组件层。

容器的轻量级性质也开辟了大量的机会，如在自动构建、发布、下载、复制等都有一席之地。使用几行命令就可以下载、构建、传输和运行容器，也可以使用 REST 风格 API 使容器对开发人员更加友好。几秒就可以构建新的容器，容器现在是自动部署流水线的一部分。

总之，容器和 VMs 各有千秋。许多组织同时使用容器和 VMs，如通过在 VM 之上运行容器。

容器的好处

我们已经考虑了容器对 VM 的许多好处。本节将解释容器超越 VM 的好处的整体优势：

- 自包含：容器将必需的应用程序二进制文件及其依赖关系打包在一起，以确保不同环境（如开发、测试和生产）之间没有差异。这促进了十二因素应用程序和不可变容器的概念。Spring Boot 微服务捆绑所有必需的应用程序依赖项。容器通过嵌入 JRE 和其他操作系统级库、配置等，进一步扩展

这个边界。

- 轻量级：容器通常非常小，占据的空间很少。最小的容器 Alpine 小于 5MB。最简单的 Spring Boot 微服务与带有 Java 8 的 Alpine 容器一起打包，大小只有约 170MB，远远小于虚拟机镜像大小（通常为 GB 级）。较小的空间不仅有助于快速启动新容器，而且使构建、传输和存储更容易。
- 可扩展性：由于容器镜像较小，并且启动时并没有涉及操作系统的启动，因此容器的启动和关闭通常很快。这使得容器成为云环境中弹性应用程序的理想选择。
- 便携式：容器提供跨机器和云提供商的可移植性。一旦使用所有依赖关系构建在容器中，就可以将它们移植到横跨多个机器或多个云环境，而不依赖于底层机器。容器可从台式机移植到不同的云环境。
- 较低的许可证成本：许多软件许可证条款基于操作系统。由于容器共享操作系统并且不在物理资源级别虚拟化，因此在许可证成本方面具有优势。
- DevOps：容器的轻量级脚本让容器本身容易自动化构建、发布和从远程存储库下载。通过与自动化交付管道集成，可以轻松地在敏捷和 DevOps 环境中使用容器。通过在构建时创建不可变容器并将它们移动到多个环境中，容器支持一次性构建的概念。由于容器不深入底层硬件，DevOps 团队可以轻易地管理容器。
- 版本控制：容器默认支持版本控制。这有助于像版本化归档文件一样，构建版本化组件。
- 可重复使用：容器镜像是可重复使用的。如果是通过组装多个库来构建镜像，就可以在类似的情况下重复使用。
- 不可变容器：在这个概念中，容器是在使用后创建和配置的，从不更新或修补。不可变容器可以避免在修补部署单元中的复杂性以及修补导致缺乏可追踪性和不一致性。



微服务和容器

微服务和容器之间没有直接的关系。微服务可以没有容器运行，容器可以运行整个应用程序。然而，二者之间隐藏着一个甜区。

容器对整体应用程序是有好处的，但是整体应用程序的复杂性和大小可能会抵消容器的一些好处。例如，使用单体应用程序时，很难快速启动新容器。除此之外，单体应用程序通常具有局部环境依赖性，如本地磁盘、与其他系统的依赖性等。这样的应用程序难以用容器技术来管理。这就是微服务与容器需要携手并进的地方。

图 8-4 显示了在同一主机上运行并共享同一操作系统，从 Contrainer SW 之上运行了 3 种语言的微服务。



图 8-4

在管理许多多语言微服务时，可以看到容器的真正优势，如用 Java、Erlang 或其他语言编写的微服务。容器帮助开发人员以技术不可知的方式来封装以任何语言或技术编写的微服务，并将它们均匀地分布在多个环境中。容器消除了使用不同的部署管理工具来处理多语言微服务的困难。它不仅抽象运行环境，还知道如何访问服务。无须考虑使用的具体技术，容器化的微服务暴露 REST 风格 API。一旦容器启动并运行，就会绑定到某些端口并暴露其 API。由于容器是自包含的，并且在

服务之间提供完全堆栈隔离，在单个 VM 或裸机中，可以运行多个异构微服务并以统一的方式处理它们。

Docker 简介

前面的章节讨论了容器及其优点。容器已经商用多年了，但是 Docker 的普及给了容器一个新的前景，因此衍生了很多新的概念。Docker 是如此受欢迎，甚至容器



化被称为 dockerization (docker 化)。

Docker 是一个构建、传输和运行基于 Linux 内核的轻量级容器的平台。Docker 默认支持 Linux 平台, 还支持使用 Boot2Docker 的 Mac 和 Windows。

Amazon EC2 容器服务 (ECS) 对 AWS EC2 实例上的 Docker 提供简单易用的支持。Docker 可以安装在裸机上, 也可以安装在传统虚拟机 (如 VMWare 或 Hyper-V) 上。

Docker 的关键组件

Docker 安装有两个关键组件: Docker 守护进程和 Docker 客户端。图 8-5 显示了 Docker 安装的关键组件。

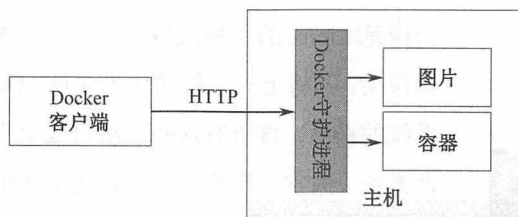


图 8-5

Docker 守护进程

Docker 守护进程是在负责构建、运行和分发 Docker 容器在主机上运行的服务器端组件。Docker 守护进程为 Docker 客户端提供了与守护进程交互的 API。这些 API 是基于 REST 的端点。Docker 守护进程可以作为在主机上运行的控制器服务。开发人员也可以编程使用这些 API 来构建自定义客户端。

Docker 客户端

Docker 客户端是一个远程命令行程序 (CLI), 通过套接字或 REST API 与 Docker 守护进程交互。CLI 可以和守护进程在同一主机上运行, 也可以在完全不同的主机上运行, 并远程连接到守护进程。Docker 用户使用 CLI 构建, 传输和运行 Docker 容器。



Docker 概念

Docker 体系结构是围绕以下几个概念构建的：镜像、容器、Registry 和 Dockerfile。

Docker 镜像

Docker 镜像是最关键的概念之一，它是操作系统库、应用程序和其库的只读副本。一旦镜像被创建，就可以无须修改而在任何 Docker 平台上运行。

在 Spring Boot 微服务中，Docker 镜像打包操作系统，如 Ubuntu、Alpine、JRE 和 Spring Boot 应用程序 jar 包。它还包括运行应用程序和服务接口的说明：

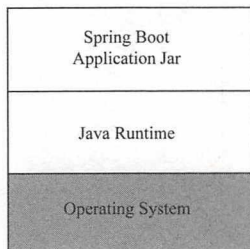


图 8-6

如图 8-6 所示，Docker 镜像基于分层架构，其中基本镜像是 Linux 的一种风格，图中每个图层都会添加到基本镜像层，并将上一个镜像作为父层。Docker 使用联合文件系统的概念，将所有这些层组合成单个镜像，形成单个文件系统。通常，开发人员无须从头开始构建 Docker 镜像。Spring 微服务中的基本镜像可以是 JRE 8，操作系统或其他通用库（如 Java 8 镜像）的镜像可从可信开源库获得，而不是从 Linux 分发镜像（如 Ubuntu）。

每次我们重新构建应用程序时，只有修改的层需要重新构建。所有中间层都被缓存，因此，如果没有任何修改，Docker 就会使用先前缓存的层并在层的顶部构建它。在一台机器上运行的具有相同类型的基本镜像的多个容器，将重用基本镜像，从而减小部署的大小。例如，在主机中，如果有多个容器以 Ubuntu 作为基本镜像运行，则它们都会重用相同的基本镜像。这也适用于发布或下载镜像的时候：

如图 8-7 所示，镜像中的第一层是一个称为 bootfs 的 boot 文件系统，类似于 Linux 内核和 boot 加载程序。boot 文件系统充当所有镜像的虚拟文件系统。

操作系统文件系统位于 boot 文件系统之上，这被称为 rootfs。根文件系统将典型的操作系统目录结构加载到容器。与 Linux 系

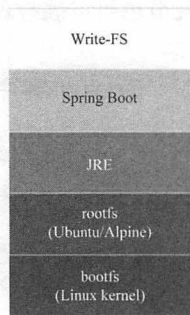


图 8-7



统不同，在 Docker 下，rootfs 是只读的。

在 rootfs 的顶部，根据需求放置其他所需的镜像，如本例中的 JRE 和 Spring Boot 微服务 jar 包。当启动容器时，将可写文件系统放在所有其他文件系统的顶部，以使进程运行。进程对底层文件系统所做的任何更改都不会反映在实际容器中。相反，这些修改被写入 volatile 的可写文件系统。因此，一旦容器停止，数据就会丢失。由于这个原因，Docker 容器本质上是非持久化的。

Docker 中打包的基本操作系统通常是只有 OS 文件系统的最小副本。实际上，顶部运行的进程可能不使用整个 OS 服务。在 Spring Boot 微服务中，容器通常只是启动 CMD 和 JVM，然后调用 Spring Boot 微服务 jar 包。

Docker 容器

Docker 容器是 Docker 镜像的运行实例。容器在运行时使用主机操作系统的内核。因此，它们与在同一主机上运行的其他容器共享主机内核。Docker 运行时通过使用内核功能（如 cgroups 和操作系统的内核命名空间），来确保进程运行在自己的隔离进程空间。除了资源防护外，容器还可以获得自己的文件系统和网络配置。

容器在被实例化时可以具有特定的资源分配，如存储器和 CPU。容器从同一镜像启动时，可以有不同的资源分配。默认情况下 Docker 容器获得一个隔离的子网和网关。网络也有 3 种模式。

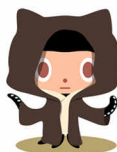
Docker Registry

Docker Registry 是发布和下载 Docker 镜像的中心位置。URL <https://hub.docker.com> 是 Docker 提供的中央 Registry。Docker Registry 具有公共镜像，它可以下载并用作基本 Registry。Docker 还有私有镜像，专用于在 Docker Registry 中创建的账户。Docker Registry 截图如图 8-8 所示。

Docker 还提供了 Docker 可信 Registry，可用于在本地设置。

Dockerfile

Dockerfile 是包含构建 Docker 镜像的指令或脚本文件。从获取基本镜像开始，Dockerfile 有多个步骤。Dockerfile 是一个通常命名为 Dockerfile 的文本文件。docker build 命令查找 Dockerfile 以获取要构建的指令。可以将 Dockerfile 与在 Maven 中的



pom.xml 文件进行类比。



图 8-8

在 Docker 中部署微服务

本节将通过展示如何为我们的 BrownField PSS 微服务构建容器来强化我们的学习。

注意本章的完整源代码在代码文件中的第 8 章项目下提供。将 chapter7.configserver、chapter7.eureka-server、chapter7.search、chapter7.search-apigateway 和 chapter7.website 复制到新的 STS 工作区中，并将其重命名为 chapter8.*。

按照下列步骤为 BrownField PSS 微服务构建 Docker 容器：

(1) 从官方 Docker 网站 <https://www.docker.com> 安装 Docker。按照“Get Started”链接，了解基于所选操作系统的下载和安装说明。安装后，使用以下命令验证安装：

```
$docker --version
Docker version 1.10.1, build 9e83765
```

(2) 在本节中，我们将了解如何 docker 化 Search 微服务(chapter8.search)，Search API 网关(chapter8.search-apigateway)微服务，以及 Website(chapter8.website) Spring Boot 应用程序。



(3) 进行修改之前, 我们需要编辑 `bootstrap.properties` 以将配置服务器 URL 从 `localhost` 更改为 IP 地址, 因为 `localhost` 在 Docker 容器中无法解析。实际操作的时候是指向 DNS 或负载均衡器, 如下所示:

```
spring.cloud.config.uri=http://192.168.0.105:8888
```

注意: 将上面的 IP 地址替换为您所使用的机器的 IP 地址。

(4) 同样, 编辑 Git 存储库上的 `search-service.properties`, 并将 `localhost` 更改为 IP 地址。这适用于 Eureka URL 和 RabbitMQ URL。更新后提交回 Git。

```
spring.application.name=search-service
spring.rabbitmq.host=192.168.0.105
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
org.springframework.cloud.config.shutdown:JFK
eureka.client.serviceUrl.defaultZone:
http://192.168.0.105:8761/eureka/
spring.cloud.stream.bindings.inventoryQ=inventory
```

(5) 修改 RabbitMQ 配置文件 `rabbitmq.config`, 取消下面内容的注释, 以提供对 `guest` 的访问。默认情况下, `guest` 仅限于从 `localhost` 访问:

```
{loopback_users, []}
```

对于不同的操作系统, `rabbitmq.config` 的位置将不同。

(6) 在 Search 微服务的根目录下创建一个 `Dockerfile`, 如下所示:

```
FROM frovlad/alpine-oraclejdk8
VOLUME /tmp
ADD target/search-1.0.jar search.jar
EXPOSE 8090
ENTRYPOINT ["java","-jar","/search.jar"]
```

以下是对 `Dockerfile` 内容的快速检查:

- `FROM frovlad/alpine-oraclejdk8`: 这告诉 Docker 构建使用特定的 `alpine-oraclejdk8` 版本作为这次构建的基本镜像。`frovlad` 指示用于定位 `alpine-oraclejdk8` 镜像的存储库。在这种情况下, 它是一个使用 Alpine Linux 和 Oracle JDK 8 构建的镜像。这样, 我们无须自己设置 Java 库, 就可以在



基本镜像之上层次化我们的应用程序。在这种情况下，由于此镜像在我们的本地镜像存储上没有，Docker 构建将继续从远程 Docker Hub 注册表下载此镜像。

- **VOLUME /tmp:** 可以从容器访问在主机中指定的目录。在我们的示例中，指向 Spring Boot 应用程序为 Tomcat 创建的工作目录的 tmp 文件夹。tmp 文件夹是容器的逻辑目录之一，间接指向主机的一个本地目录。
- **ADD target/search-1.0.jar search.jar:** 将应用程序二进制文件添加到指定了目标文件名的容器。在这种情况下，Docker 构建将 target/search-1.0.jar 作为 search.jar 拷贝到容器。
- **EXPOSE 8090:** 告诉容器如何做端口映射。将 8090 与用于内部 Spring Boot 服务的外部端口绑定。
- **ENTRYPOINT ["java","-jar","/search.jar"]:** 告诉容器，启动容器时运行哪个默认应用程序。指向的是 Java 进程和 Spring Boot 微服务 jar 包来启动。

(7) 下一步是从存储 Dockerfile 的文件夹运行 docker build，这一步是下载基本镜像并从 Dockerfile 中逐行运行，如下所示：

```
docker build -t search:1.0 .
```

该命令的输出如图 8-9 所示。

```
rvslab:chapter8.search rajeshrv$ docker build -t search:1.0
Sending build context to Docker daemon 48.34 MB
Step 1 : FROM frovlvlad/alpine-oraclejdk8
--> 5b8d90632c89
Step 2 : VOLUME /tmp
--> Using cache
--> c79a1b3275d4
Step 3 : ADD target/search-1.0.jar app.jar
--> 7766e630f139
Removing intermediate container f2ac976e781d
Step 4 : EXPOSE 8090
--> Running in 730300fa66a9
--> e058cc1615da
Removing intermediate container 730300fa66a9
Step 5 : ENTRYPOINT java -jar /app.jar
--> Running in b79116f3e54b
--> 5a8d0d6e0bf7
Removing intermediate container b79116f3e54b
Successfully built 5a8d0d6e0bf7
rvslab:chapter8.search rajeshrv$
```

图 8-9



(8) 对 Search API Gateway 和 Website 重复相同的步骤。

(9) 创建镜像后，可以通过键入以下命令进行验证。此命令将列出镜像及其详细信息，包括镜像文件的大小：

```
docker images
```

输出如图 8-10 所示。

```
rvslab:chapter8 rajeshrv$ docker images
REPOSITORY          TAG                 IMAGE ID
website              1.0                263605f253f3
search-apigateway    1.0                322890ae3ec1
search               1.0                f094b72d1b41
```

图 8-10

(10) 接下来要做的是运行 Docker 容器，命令是 `docker run`。此命令将加载和运行容器。在启动时，容器调用 Spring Boot 可执行 jar 包来启动微服务。在启动容器之前，请确保 Config 和 Eureka 服务器正在运行：

```
docker run --net host -p 8090:8090 -t search:1.0
docker run --net host -p 8095:8095 -t search-apigateway:1.0
docker run --net host -p 8001:8001 -t website:1.0
```

上述命令启动 Search、Search API Gateway 和 Website。

在本例中，我们使用主机网络（`--net 主机`）而不是桥接网络，以避免 Eureka 使用 Docker 容器名称注册。可以通过覆盖 `EurekaInstanceConfigBean` 文件来修正。从网络角度来看，与桥接网络相比，主机网络较少隔离。主机与桥接网络的优缺点取决于项目的具体情况。

(11) 所有服务完全启动后，使用 `docker ps` 命令验证，如图 8-11 所示。

```
rvslab:chapter8 rajeshrv$ docker ps
CONTAINER ID        IMAGE               COMMAND
32af53b56945        website:1.0        "java -jar /website.j"
ce35355aea65        search-apigateway:1.0 "java -jar /search-ap"
5577c5107fc6        search:1.0         "java -jar /search.ja"
4a423d0872b5        registry:2         "/bin/registry /etc/d"
rvslab:chapter8 rajeshrv$
```

图 8-11

(12) 下一步是访问 `http://192.168.99.100:8001` 的 BrownField PSS 的网址。注意上

面链接的 IP 地址。如果您在 Mac 或 Windows 上使用 Boot2Docker 运行,这是 Docker 机器的 IP 地址。在 Mac 或 Windows 中,如果 IP 地址未知,请键入以下命令以查找默认计算机的 DockerIP 地址:

```
docker-machine ip default
```

如果 Docker 在 Linux 上运行,那么这里的链接是主机 IP 地址。对 Booking、Fares、Check-in 及他们各自的网关微服务应用做相同的更改。

在 Docker 上运行 RabbitMQ

由于我们的示例还使用 RabbitMQ,接下来探讨如何将 RabbitMQ 设置为 Docker 容器。以下命令从 Docker Hub 中提取 RabbitMQ 镜像并启动 RabbitMQ:

```
docker run --net host rabbitmq3
```

确保* -service.properties 中的 URL 更改为 Docker 主机的 IP 地址。获取 IP 地址的方法上一节最后介绍过。

使用 Docker Registry

Docker Hub 提供了一个中央位置来存储所有 Docker 镜像。镜像可以存储为公共和私有的。在许多情况下,由于安全问题,在办公场所部署自己的私有注册表。

设置和运行本地注册表,需要下列步骤:

(1) 启动注册表,并在端口 5000 上绑定注册表:

```
docker run -d -p 5000:5000 --restart=always --name registryregistry:2
```

(2) 标签搜索: 1.0 到注册表,如下:

```
docker tag search:1.0 localhost:5000/search:1.0
```

(3) 然后,通过以下命令将镜像推送到注册表:

```
docker push localhost:5000/search:1.0
```

(4) 从注册表中拉取镜像:

```
docker pull localhost:5000/search:1.0
```

设置 Docker Hub

第 7 章中我们使用了本地 Docker Registry。本节将介绍如何设置和使用 Docker Hub 来发布 Docker 容器,这是全局访问 Docker 镜像的好办法。在本章的后面,将从本地机器发布 Docker 镜像到 Docker Hub,并从 EC2 实例进行下载。

创建一个公共 Docker Hub 账户和库。对于 Mac,请访问这个 URL: https://docs.docker.com/mac/step_five/。

在本示例中,使用 brownfield 为用户名创建 Docker Hub 账户。这里 Registry 充当微服务存储库,所有的 docker 化微服务在这个库被存储和访问。这是微服务功能模型中的功能之一。

将微服务发布到 Docker Hub

为了将 docker 化服务推送到 Docker Hub,需要进行下列操作。第一个命令标记 Docker 镜像,第二个命令将 Docker 镜像推送到 Docker Hub 存储库:

```
docker tag search:1.0brownfield/search:1.0
docker push brownfield/search:1.0
```

要验证容器镜像是否已发布,请访问 <https://hub.docker.com/u/brownfield> 上的 Docker Hub 存储库。对所有其他 BrownField 微服务,以此类推。完成之后,所有服务都将发布到 Docker Hub。

云上的微服务

微服务功能模型中提到的一个功能是使用云基础设施来实现微服务。在本章的前面部分,我们还探讨了使用云进行微服务部署的必要性。到目前为止,我们还没有在云上部署任何东西。因为在我们的整个 BrownField PSS 微服务生态系统中总体上有 8 个微服务: Config、Eureka、Turbine、RabbitMQ、Elasticsearch、Kibana 和

Logstash，所以很难在本地机器上运行它们。

在本书的其余部分，我们将使用 AWS 作为云平台来部署 BrownField PSS 微服务。

在 AWS EC2 上安装 Docker

本节中，我们将在 EC2 实例上安装 Docker。前提是假设读者熟悉 AWS 并且已经在 AWS 上创建了一个账户。在 EC2 上设置 Docker，需要下列步骤：

(1) 启动新的 EC2 实例。如果我们必须同时运行所有服务，可能需要一个较大的 EC2 实例。示例使用 `t2.large`。在本例中，使用下面的 Ubuntu AMI 镜像：`ubuntu-trusty-14.04-amd64-server-20160114.5` (`ami-fce3c696`)。

(2) 连接到 EC2 实例并运行以下命令：

```
sudo apt-get update
sudo apt-get install docker.io
```

(3) 上述命令将在 EC2 实例上安装 Docker。

验证是否安装成功需要使用以下命令：

```
docker version
```

在 EC2 上运行 BrownField 服务

本节我们将在创建的 EC2 实例上设置 BrownField 微服务。在这种情况下，我们将在本地桌面设置构建过程和在 AWS 上部署二进制文件，并遵循以下步骤：

(1) 命令安装 Git：

```
sudo apt-get install git
```

(2) 在任意文件夹上创建一个 Git 存储库。

(3) 修改 Config server 的 `bootstrap.properties`，指向为此示例上一步 Git 存储库。

(4) 使用 EC2 实例的私有 IP 地址修改所有微服务的 `bootstrap.properties`，指向 `config-server`。

(5) 将所有 `*.properties` 从本地 Git 存储库复制到 EC2 Git 存储库并提交。

(6) 更改 `*.properties` 文件中的 Eureka 服务器 URL 和 RabbitMQ URL 以匹配 EC2 私有 IP 地址。完成后将更改提交到 Git。

(7) 在本地机器上，重新编译所有项目，并为 `search`、`Search API Gateway` 和 `website` 微服务创建 Docker 镜像，并将它们都推送到 Docker Hub。

(8) 从本地计算机复制 `config-server` 和 `Eureka-server` 二进制文件到 EC2 实例。

(9) 在 EC2 实例上设置 Java 8。

(10) 然后，按顺序执行以下命令：

```
java -jar config-server.jar
java -jar eureka-server.jar
docker run --net host rabbitmq:3
docker run --net host -p 8090:8090 rajeshrv/search:1.0
docker run --net host -p 8095:8095 rajeshrv/searchapigateway:
1.0
docker run --net host -p 8001:8001 rajeshrv/website:1.0
```

(11) 打开网站的 URL 并执行搜索来检查所有服务是否正常。注意：我们要使用公共 IP 地址：`http://54.165.128.23:8001`。

更新生命周期管理器

在第 6 章“自动化扩（缩）容微服务”中，我们使用生命周期管理器来自动启动和停止实例。我们使用 SSH 并执行一个 Unix 脚本来启动目标机器上的 Spring Boot 微服务。现在有了 Docker，我们不再需要 SSH 连接，因为 Docker 守护进程提供了基于 REST 的 API 来启动和停止实例。这极大地简化了生命周期管理器部署的复杂性。

本节中，我们不会重写生命周期管理器。总的来说，我们将在第 9 章中替换整个生命周期管理器。

容器化的未来——内核和强化安全

容器化仍在发展，但是最近采用容器化技术的组织越来越多。虽然许多组织积极采用 Docker 和其他容器技术，但仍然面临着容器的大小和安全问题。

目前，Docker 镜像很重。在容器频繁创建和销毁的弹性自动化环境中，大小仍然是一个问题。较大的容器意味着更多的代码，更多的代码意味着它更容易出现安全漏洞。

未来属于小容器。Docker 正在开发单核、轻量级内核，即使在低功耗物联网设备上也可以运行 Docker。Unikernels 不是成熟的操作系统，但它们提供了支持部署的应用程序所必须的库。

容器的安全问题引发了很多争论。关键的安全问题是围绕用户命名空间或用户 ID 隔离。如果容器在 root 上，那么它可以默认获得主机的 root 权限。另一个安全问题是使用来自不受信任来源的容器镜像。Docker 尽可能快地缩小这些差距，但是有很多组织使用 VM 和 Docker 的组合来规避一些安全问题。

总结

在本章中，我们了解了在处理互联网级微服务时云环境的必要性。

我们探索了容器的概念，并将其与传统虚拟机进行了比较。学习了 Docker 的基础知识，解释了 Docker 镜像、容器和 Registry 的概念。紧接着在微服务的上下文中展示了容器的好处。

然后，本章通过 Docker 化 BrownField 微服务的实例，演示了如何在 Docker 上部署之前开发的 Spring Boot 微服务。探索了如何通过本地 Registry 及 Docker Hub 来推送和拉取 docker 化的微服务。

最后，我们探讨了如何在 AWS 云环境中部署一个 Docker 化的 BrownField 微服务。

第 9 章

使用 Mesos 和 Marathon 管理 Dockerized 微服务



在互联网规模的微服务部署中，管理数千个 docker 化微服务并不容易。必须拥有基础设施抽象层和强大的集群控制平台，以便管理。

本章将分别介绍 Mesos 和 Marathon 作为基础设施抽象层和集群控制系统的必要性和使用，以便在大规模部署微服务的云环境中实现资源的优化使用。本章还将介绍在云环境中设置 Mesos 和 Marathon 的方法。最后，演示如何在 Mesos 和 Marathon 环境中管理 docker 化微服务。

学习完本章，您将了解到：

- 抽象层和集群控制软件的必要性。
- 从微服务的上下文角度看 Mesos 和 Marathon。
- 使用 Mesos 和 Marathon 管理 BrownField PSS 的 Docker 化微服务。

回顾微服务功能模型

我们将从第 3 章“微服务概念的应用”中讨论的微服务功能模型入手，探索集群的控制和配置功能，如图 9-1 所示。

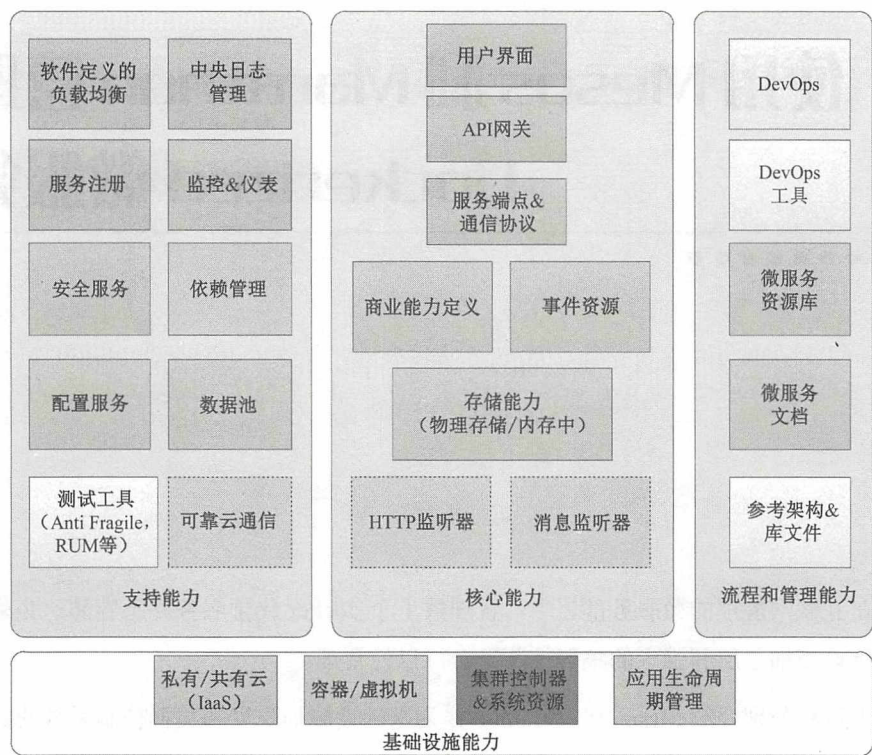


图 9-1

缺少的部分

在第 8 章“用 Docker 实现容器化微服务”中，我们讨论了如何将 BrownField 航空的 PSS 微服务 docker 化。Docker 将 JVM 运行、OS 参数和应用程序一起打包，

以便 Docker 化微服务可以跨环境工作。Docker 提供的 REST API 简化了生命周期管理器在启动和停止时与目标机器的交互过程。

在大规模部署中，有数以千计的 Docker 容器，我们需要确保 Docker 容器运行自己的资源，如内存、CPU 等。除此之外，可能有为 Docker 部署设置的规则，如容器的副本不应该在同一台机器上运行等。此外，需要一个机制来优化服务器基础设施的使用，以避免产生额外的成本。

有些组织处理数十亿个容器。手动管理是不可能的。在大型 Docker 部署的背景下，需要回答一些关键问题：

- 如何管理数千个容器？
- 如何监控？
- 在部署应用时如何设定规则和约束？
- 如何确保我们正确使用容器获得资源？
- 如何确保在任何时间点至少有一定数量的最小实例正在运行？
- 如何确保依赖服务启动并运行？
- 如何进行滚动升级和优雅的迁移？
- 如何回滚故障部署？

所有这些问题的解决都需要两个关键功能：

- 在许多物理或虚拟机上提供统一抽象的集群抽象层。
- 集群控制和初始化系统，用于在集群抽象上智能地管理部署过程。

生命周期管理器能理想地处理这些情况，我们可以在上面添加一些功能来解决这些问题。然而，在尝试修改生命周期管理器之前，重要的是要了解集群管理解决方案的作用。

为什么集群管理很重要

由于微服务将应用程序分成不同的微应用程序，开发人员需要更多的服务器节

点进行部署。为了正确地管理微服务，通常每个虚拟机部署一个微服务，这进一步降低了资源利用率，因为可能导致 CPU 和内存的过度分配。

许多部署中，微服务的高可用性要求迫使工程师添加越来越多的服务实例来实现冗余。一方面虽然带来了高可用性，却有可能导致一些服务器实例未充分利用。

一般来说，与单一应用程序部署相比，微服务部署需要更多的基础架构。由于基础架构带来的成本增加，抵消了微服务本身带来的一些价值，如图 9-2 所示。

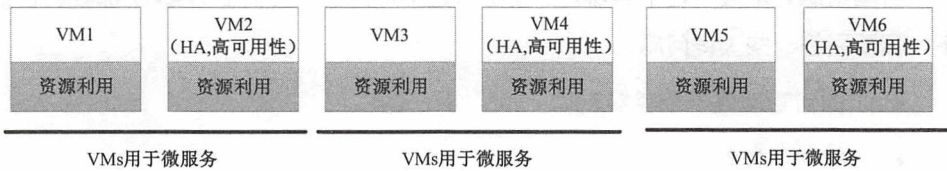


图 9-2

为了解决之前提到的问题，我们需要一个能够满足以下要求的工具：

- 自动化，如有效地将容器分配给基础架构，并使开发人员和管理员保持透明。
- 为开发人员提供一个抽象层，以便他们可以针对数据中心部署其应用程序，而无须搞清楚是部署在哪台机器上。
- 针对部署组件设置规则或约束。
- 提供更高的敏捷性、更少的管理开销和沟通难度。
- 最大限度地利用可用资源，来构建、部署和管理应用程序。

我们选择的任何具备上述功能的工具，都可以统一的方式处理容器，而无须关心底层的微服务技术。

集群管理能做什么

典型的集群管理工具可以虚拟化一组计算机，并将其作为单个集群进行管理。集群管理工具可以透明化地跨机器移动负载或容器，集群管理包括集群编排、集群管理、数据中心虚拟化、容器调度、容器生命周期管理、容器编排、数据中心等。

许多工具目前支持基于 Docker 的容器及非容器化的二进制应用进行部署，如独立的 Spring Boot 应用程序。这些集群管理工具的基本功能是为应用程序开发人员和管理员抽象管理物理服务器。

集群管理工具有助于基础架构的自助服务和配置，无须基础架构团队使用预定义的规范分配所需机器。在这种自动化集群管理方法中，机器不再预先配置或分配给应用程序。一些集群管理工具还可以跨多台机器甚至跨数据中心，进行数据中心的虚拟化，并创建一个弹性的、类似私有云的基础架构。集群管理工具没有标准参考模型。因此，不同的服务提供商的功能不同。

集群管理软件的关键功能总结如下：

- 集群管理：将一组虚拟机和物理机作为单个大型计算机管理。这些机器在资源能力方面虽然不同，但都是以 Linux 作为操作系统的机器。这些虚拟集群可以是云、内部或两者的组合。
- 部署：使用大量机器来处理应用程序和容器的自动部署，并且支持多个版本的应用程序容器，以及跨大量集群机器的滚动升级。这些工具还能够处理故障的回滚。
- 可扩展性：在需要的时候，处理应用程序实例的自动和手动伸缩性，以优化利用率为主要目标。
- 运行状况：管理集群、节点和应用程序的运行状况。从集群中删除故障的机器和应用程序实例。
- 基础架构抽象：将开发人员从部署应用程序的机器上抽象化。开发人员不必关心机器及其容量，而是完全由集群管理软件决定如何计划和运行应用程序。这些工具还从开发人员抽象机器详细信息、容量、利用率和位置。对于应用程序所有者，相当于具有无限容量的单台大型机器。
- 资源优化：这些工具的固有行为是在一组可用的机器上，利用算法有效地分配容器工作负载，从而降低成本。
- 资源分配：基于开发人员设置的约束、关联性规则、端口要求、应用程序依赖性、运行状况等进行资源的分配。

- 服务可用性：它确保服务在集群中启动并运行。机器故障的时候，集群控制工具会在集群中的其他机器上重新启动这些服务。
- 敏捷：如果资源需求有变化，这些工具能够快速地将工作负载分配给可用资源，或者跨机器移动工作负载。并且可以设置约束，基于业务的关键性、优先级等重新调整资源。
- 隔离：一些工具提供了简单易用的资源隔离。因此即使应用不是容器化的，也可以实现资源隔离。

资源分配有多种算法，从简单算法到复杂算法包括机器学习和人工智能。常见算法是随机、二进制打包和扩展。应用程序设置的约束将覆盖基于资源可用性的默认算法。

图 9-3 用两台机器显示了这些算法如何使用部署填充可用的计算机：

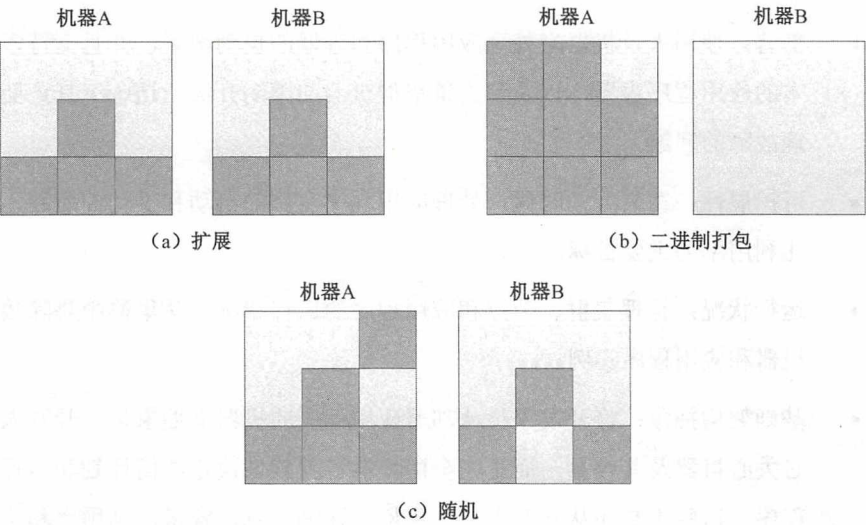


图 9-3

- 扩展：此算法在可用计算机上平均分配工作负载，如图（a）所示。
- 二进制打包：该算法尝试通过机器填充数据，并确保机器的最大利用率。当以按用户付费的方式使用云服务时，二进制打包质量很高，如图（b）所示。

- 随机：随机选择机器并部署容器，如图（c）所示。

还有使用认知计算算法（如机器学习和协作过滤）以提高效率的可能性。诸如超额预订的技术通过为高优先级任务（如分析、视频、镜像处理等的创收服务）分配未充分利用的资源。

与微服务的关系

微服务的基础架构，如果没有正确配置，很容易导致架构冗余和更高的成本。如前所述，使用集群管理工具的云环境对于处理大规模微服务的成本效益至关重要。

使用 Spring Cloud 项目进行的 Spring Boot 微服务，能够无缝运用到各种集群管理工具（大部分工具都是兼容 Spring 的产品）。由于基于 Spring Cloud 的微服务的位置是不可知的，这些服务可以部署在集群中的任何地方。每当服务启动时，都会自动注册到服务注册表并宣布其可用性。另外，消费者通过注册中心来发现可用的服务实例。应用程序支持完整的流体结构，而无须预先部署拓扑。使用 Docker，我们能够运行在一个虚拟的环境中，以便服务可以在任何基于 Linux 的环境上跨平台移植和运行。

与虚拟化的关系

集群管理解决方案不同于服务器虚拟化解决方案。作为应用程序组件，集群管理解决方案在 VM 或物理机之上运行。

集群管理解决方案

有许多可用的集群管理软件工具。在它们之间做比较是不公平的。即使没有一

对一的组件，这些工具之间的能力有很多重叠的功能。在许多情况下，组织使用单个或多个这些工具的组合来满足他们的要求。

图 9-4 显示了微服务上下文中集群管理工具的位置。

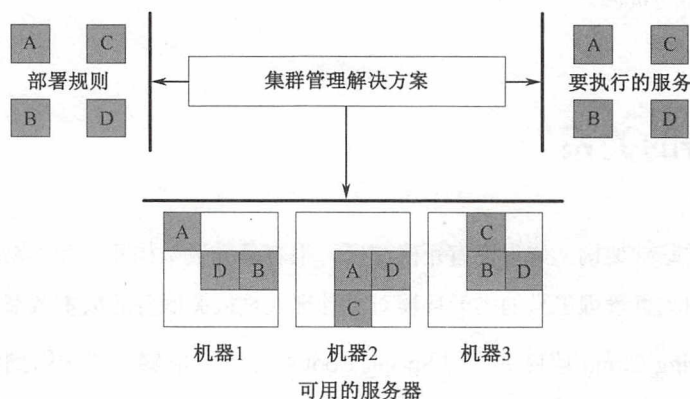


图 9-4

在本节中，我们将探讨一些广泛使用的集群管理解决方案。

Docker Swarm

Docker Swarm 是 Docker 的本地集群管理解决方案。Swarm 提供了与 Docker 的更深入集成，并暴露了与 Docker 的远程 API。Docker Swarm 在逻辑上将 Docker 主机池分组，并将其作为一个大型 Docker 虚拟主机管理。应用程序管理员和开发人员无须考虑将容器部署在哪个主机上，一切都交给 Docker Swarm。Docker Swarm 将根据 bin 打包和扩展算法决定使用哪个主机。

由于 Docker Swarm 基于 Docker 的远程 API，因此，与其他任何容器编排工具相比，Docker Swarm 是市场上一个相对较新的产品，它只支持 Docker 容器。

Docker Swarm 使用管理器和节点的概念，管理器是主管交互和编排 Docker 容器，节点是部署和运行 Docker 容器的位置。

Kubernetes

Kubernetes (k8s) 来自 Google 的工程，用 Go 语言编写，在 Google 上进行大规模

模部署时经过了测试。与 Swarm 类似，Kubernetes 帮助跨集群节点管理容器化应用程序。Kubernetes 可以实现自动化容器部署、调度和容器的可扩展性。Kubernetes 支持一些开箱即用的有用功能，如自动渐进式部署、版本化部署和容器弹性（如果容器由于某种原因失败）。

Kubernetes 架构具有 master、nodes、pods 的概念。Master 和节点一起形成 Kubernetes 集群。Master 负责跨多个节点分配和管理工作负载。节点只是 VM 或物理机，节点进一步被子段化为 Pods，节点可以托管多个 Pods，一个或多个容器被分组并在 Pods 中执行。Pods 还有助于管理和部署协同定位的服务以提高效率。Kubernetes 还支持标签的概念作为查询和查找容器的键值对。标签是用户定义的参数，用于标记执行常见类型工作的节点，如前端 Web 服务器。部署在集群上的服务获取单个 IP/DNS 以访问服务。

Kubernetes 对 Docker 提供了开箱即用的支持，然而 Kubernetes 学习难度比 Docker Swarm 更高。RedHat 为 Kubernetes 提供商业支持，作为其 OpenShift 平台的一部分。

Apache Mesos

Mesos 是一个开源框架，最初由加州大学伯克利分校开发，被 Twitter 大规模使用。Twitter 使用 Mesos 主要管理大型 Hadoop 生态系统。

Mesos 与以前的解决方案略有不同。Mesos 更多的是一个资源管理器，它依靠其他框架来管理工作负载执行。Mesos 位于操作系统和应用程序之间，提供了一个逻辑集群的机器。

Mesos 是一个分布式系统内核，逻辑上将多台计算机分组并虚拟到单个大型机器。Mesos 能够将多个异构资源分组到可以部署应用程序的统一资源集群。由于这些原因，Mesos 也被称为在数据中心中构建私有云的工具。

Mesos 具有 Master 和 slave 的概念。与之前的解决方案类似，Master 负责管理集群，而从属节点运行工作负载。Mesos 内部使用 ZooKeeper 进行集群协调和存储。Mesos 支持框架的概念。这些框架负责调度和运行非容器应用程序和容器。Marathon、Chronos 和 Aurora 是用于调度和执行应用程序的流行框架。Netflix Fenzo 是另一个开源的 Mesos 框架。有趣的是 Kubernetes 也可以用作 Mesos 框架。



Marathon 支持 Docker 容器及非容器应用程序。Spring Boot 可以直接在 Marathon 中配置。Marathon 提供了一些开箱即用的功能，如支持应用程序的依赖关系，将应用程序分组为扩展和升级服务，启动和关闭健康和异常的实例，推出升级，回滚失败升级等。

Mesosphere 为 Mesos 和 Marathon 提供商业支持，并把它们作为其 DCOS 平台的一部分。

Nomad

来自 HashiCorp 的 Nomad 是另一个集群管理软件。Nomad 是一个集群管理系统，用于抽象较低级别的机器详细信息及其位置。与之前探讨的其他解决方案相比，Nomad 具有更简单的架构，Nomad 也是轻量级的。与其他集群管理解决方案类似，Nomad 负责资源分配和应用程序的执行。Nomad 还接受用户特定的约束并基于此分配资源。

Nomad 具有服务器的概念，所有作业都在其中进行管理。一个服务器充当领导者，而其他服务器充当跟随者。Nomad 有任务的概念，这是最小的工作单元。任务被分组到任务组中。任务组具有要在相同位置执行的任务。一个或多个任务组或任务作为作业进行管理。

Nomad 支持许多工作负载，包括 Docker 开箱即用。Nomad 还支持跨数据中心的部署。

Fleet

Fleet 是 CoreOS 的集群管理系统。它运行在较低级别，并在 systemd 的顶部工作。Fleet 可以管理应用程序依赖关系，并确保所有所需的服务都在集群中的某处运行。如果服务失败，它将在另一台主机上重新启动该服务。在分配资源时可以提供相关性和约束规则。

Fleet 有引擎和代理的概念。在集群中的任何点处只有一个引擎具有多个代理。任务提交到引擎，代理在集群计算机上运行这些任务。

Fleet 还支持 Docker 开箱即用。



集群管理与 Mesos 和 Marathon

正如我们在上一节中讨论的，有许多集群管理解决方案或容器编排工具可用。不同的组织选择不同的解决方案来解决基于环境的问题。许多组织选择一个支持 Kubernetes 或 Mesos 的框架，如 Marathon。在大多数情况下，Docker 用作默认的容器化方法来打包和部署工作负载。

在本章的剩余部分，我们将展示 Mesos 如何与 Marathon 合作，提供所需的集群管理功能。许多组织都在使用 Mesos，包括 Twitter、Airbnb、Apple、eBay、Netflix、PayPal、Uber、Yelp 等。

深入 Mesos

Mesos 可以被视为数据中心内核。DCOS 是 Mesosphere 支持的 Mesos 的商业版本。为了在一个节点上运行多个任务，Mesos 使用资源隔离概念。Mesos 依靠 Linux 内核的 cgroups 来实现类似于容器方法的资源隔离。它还支持使用 Docker 的容器化隔离。Mesos 支持批处理工作负载以及 OLTP 类工作负载，如图 9-5 所示。

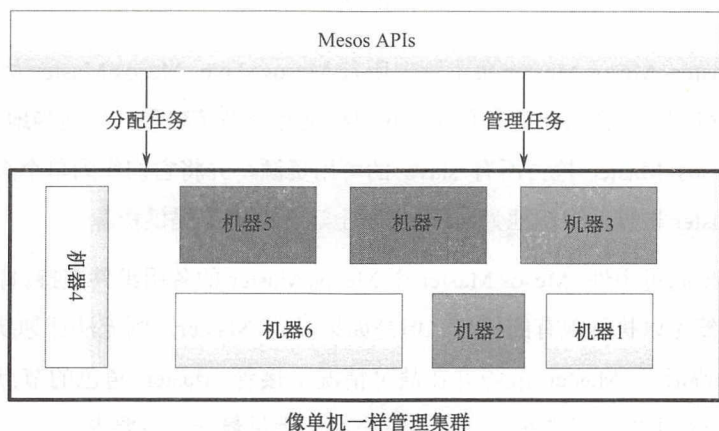


图 9-5

Mesos 是 Apache 许可证下的开源顶级 Apache 项目。Mesos 从较低级别的物理



或虚拟机中抽象较低级别的计算资源，如 CPU、内存和存储。

在研究为什么我们需要 Mesos 和 Marathon 之前，需要先了解 Mesos 架构。

Mesos 架构

图 9-6 展示了简单的 Mesos 架构。Mesos 的关键组件包括一个 Mesos Master、一组 slave、一个 ZooKeeper 服务和一个 Mesos 框架。Mesos 框架进一步细分为两个组件：调度程序和执行程序。

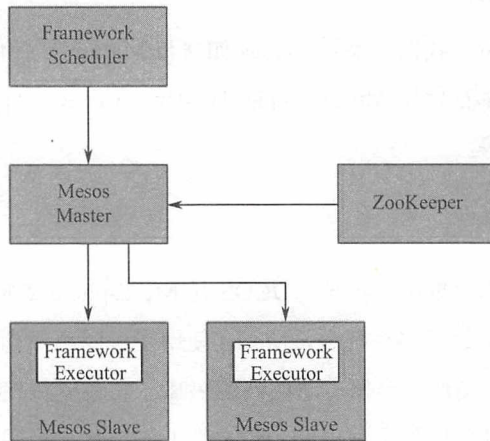


图 9-6

- **Master:** Mesos Master 负责管理所有 Mesos slave。Mesos Master 从所有 slave 获得有关资源可用性的信息，并根据某些资源策略和约束适当地填充资源。Mesos Master 抢占所有 slave 的可用资源，并将它们作为单个大机器池。Master 根据此资源池为 slave 机器上运行的框架提供资源。

为了实现高可用性，Mesos Master 由 Mesos Master 的备用组件支持。即使 Master 不可用，仍然可以执行现有的任务。但是如果缺少 Master，则无法计划新任务。主备节点是等待活跃 Master 故障并在故障情况下接管 Master 角色的节点，它使用 ZooKeeper 进行主领导者选举。领导者选举必须满足最低人数要求。

- **Slave:** Mesos slave 负责托管任务执行框架。任务在 slave 上执行。
- **ZooKeeper:** ZooKeeper 是一个集中协调服务器，用于 Mesos 协调 Mesos 集

群上的活动。Mesos 在 Mesos Master 故障的情况下使用 ZooKeeper 进行领导选举。

- **Framework:** Mesos 框架负责理解应用程序的约束，接受来自主机的资源提供，最后在 Master 提供的 slave 资源上运行任务。Mesos 框架由两个组件组成：框架调度器（framework scheduler）和框架执行器（framework executor）。
 - 框架调度程序（framework scheduler）：调度程序负责注册到 Mesos 和处理资源提供。
 - 框架执行器（framework executor）：执行器在 Mesos slave 上运行实际的程序。

该框架还负责实施某些策略和约束。例如，执行时至少保证 500MB 的 RAM。框架是可插入组件，可以用另一个框架替换，框架工作流程如图 9-7 所示。

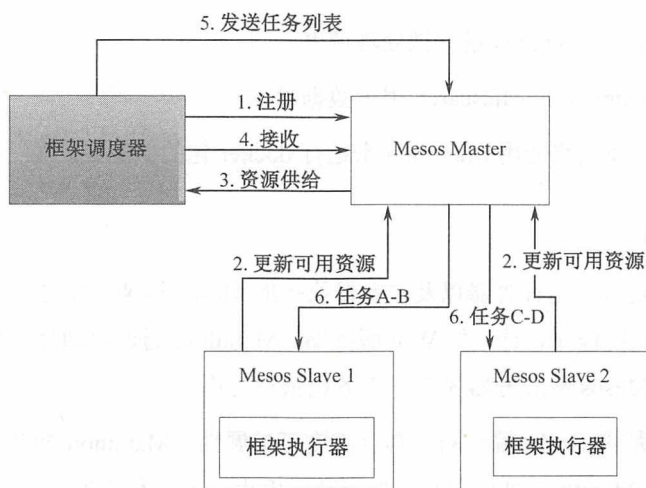


图 9-7

图 9-7 表示的流程如下。

(1) 框架向 Mesos 主机注册，并等待资源提供。调度器可以用不同的资源（在该示例中为任务 A~D）来分别执行任务。

(2) Mesos slave 为 Mesos Master 提供可用的资源。例如，slave 通告自身的可用 CPU 和内存。

(3) Mesos Master 根据分配策略集创建资源建议，将其提供给框架的调度程序组件。分配策略确定将资源提供给哪个框架及要提供多少资源。可以通过插入其他分配策略来自定义默认策略。

(4) 调度器框架组件基于约束、能力和策略，可以接受或拒绝资源提供。例如，如果根据所设置的约束和策略资源不足，则框架拒绝资源提供。

(5) 如果调度程序组件接受资源提议，则它向 Mesos Master 提交一个任务的详细信息，每个任务具有资源约束。在这个例子中，它准备提交任务 A~D。

(6) Mesos Master 将此任务列表发送到资源可用的 slave。安装在 slave 机器上的框架执行器组件将会运行这些任务。

Mesos 支持多种框架，如下：

- Marathon 和 Aurora 用于长时间运行的进程，如 Web 应用程序。
- Hadoop、Spark 和 Storm 进行大数据处理。
- Chronos 和 Jenkins 进行批处理调度。
- Cassandra 和 Elasticsearch 用于数据管理。

在本章中，我们将使用 Marathon 来运行 docker 化的微服务。

Marathon

Marathon 是可以运行容器以及非容器执行的 Mesos 框架实现之一。Marathon 专为长期运行的应用程序设计，如 Web 服务器。Marathon 通过启动另一个实例来确保即使它托管的 Mesos 从服务器失败，服务也依然可用。

Marathon 是用 Scala 编写的，具有高度可扩展性。Marathon 提供了一个 UI 及 REST API 来与 Marathon 进行交互，如启动、停止、扩展和监视应用程序。

与 Mesos 类似，Marathon 的高可用性是通过运行多个注册到 ZooKeeper 的 Marathon 实例来实现的。一个 Marathon 实例充当领导者，而其他实例处于待机模式。如果主导主失败，则将进行领导者选择，并且将确定下一个活动主控器。

Marathon 的一些基本特征包括：

- 设置资源约束。
- 扩展、缩小和应用程序的实例管理。

- 应用程序版本管理。
- 启动和杀死应用程序。

Marathon 的一些高级功能包括：

- 滚动升级、滚动重新启动和回滚。
- 蓝绿部署。

为 BrownField 微服务实现 Mesos 和 Marathon

在本节中，将把第 8 章“用 Docker 实现容器化微服务”部署的容器化的 Brownfield 微服务部署到 AWS 云中，并使用 Mesos 和 Marathon 进行管理。

为了演示的目的，在解释中仅涵盖 3 个服务（搜索、搜索 API 网关和网站）。

逻辑架构如图 9-8 所示。该实现使用多个 Mesos slave，使用单个 Mesos Master 执行 docker 化的微服务。Marathon 调度程序组件用于调度这些微服务。Docker 化微服务已经托管在 Docker Hub 注册表上，它们是使用 Spring Boot 和 Spring Cloud 实现的。

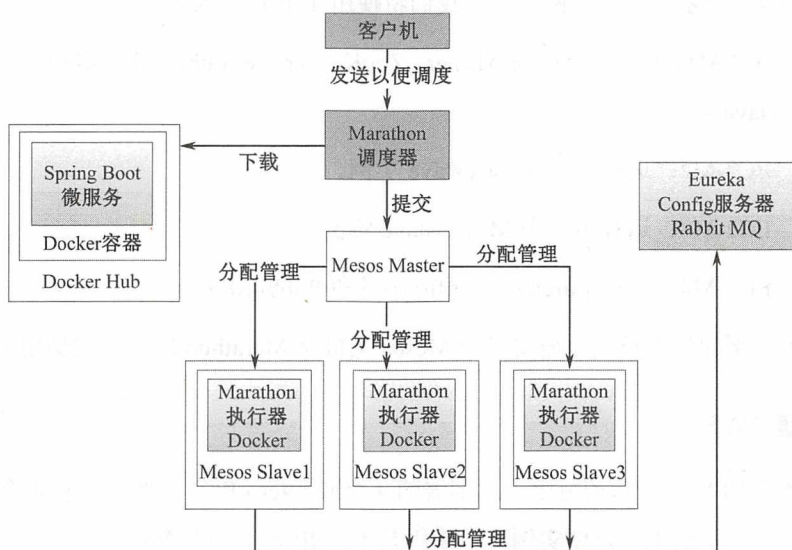


图 9-8

图 9-9 显示了物理部署体系结构。

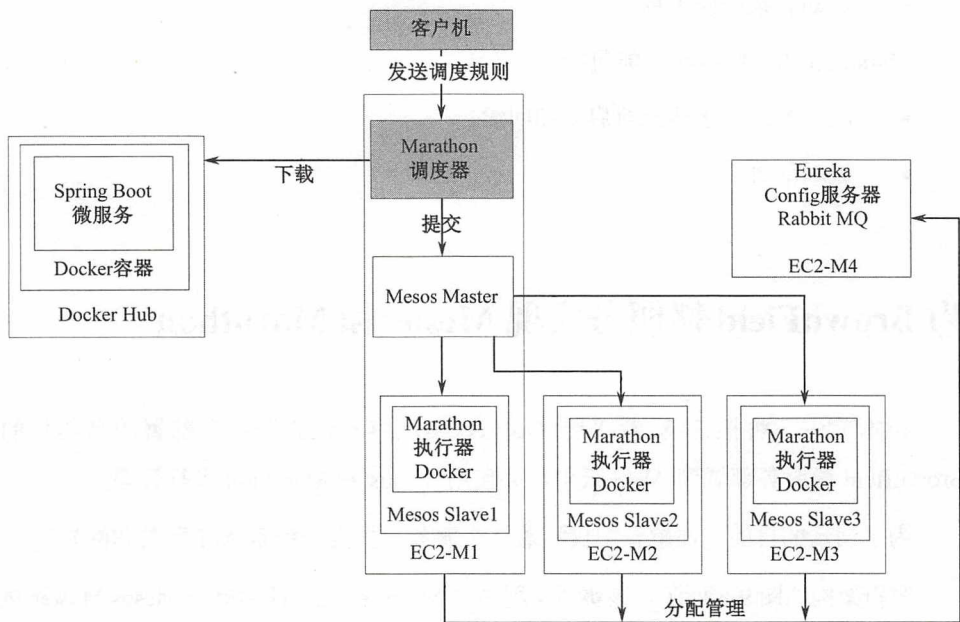


图 9-9

如图 9-9 所示，在这个例子中，我们将使用 4 个 EC2 实例：

- EC2-M1：运行了 Mesos Master、ZooKeeper、Marathon 调度器和一个 Mesos slave 实例。
- EC2-M2：运行一个 Mesos slave 实例。
- EC2-M3：运行另一个 Mesos slave 实例。
- EC2-M4：运行 Eureka、Config 服务和 RabbitMQ。

对于真正的生产设置，需要多个 Mesos 主机及 Marathon 的多个实例用于容错。

设置 AWS

如图 9-10 所示，启动将用于此部署的 4 个 t2.micro EC2 实例。所有 4 个实例必须在同一个安全组中，以便实例可以使用其本地 IP 地址相互查看。

图 9-11 显示了机器详细信息和 IP 地址。

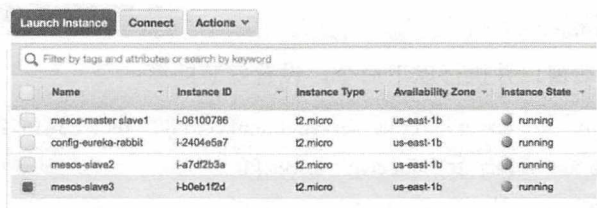


图 9-10

Instance ID	Private DNS/IP	Public DNS/IP
i-06100786	ip-172-31-54-69.ec2.internal 172.31.54.69	ec2-54-85-107-37.compute-1.amazonaws.com 54.85.107.37
i-2404e5a7	ip-172-31-62-44.ec2.internal 172.31.62.44	ec2-52-205-251-150.compute-1.amazonaws.com 52.205.251.150
i-a7df2b3a	ip-172-31-49-55.ec2.internal 172.31.49.55	ec2-54-172-213-51.compute-1.amazonaws.com 54.172.213.51
i-b0eb1f2d	ip-172-31-53-109.ec2.internal 172.31.53.109	ec2-54-86-31-240.compute-1.amazonaws.com 54.86.31.240

图 9-11

根据您的 AWS EC2 配置替换 IP 和 DNS 地址，安装 ZooKeeper、Mesos 和 Marathon。

以下软件版本将用于部署。本节中的部署遵循前面部分的物理部署架构：

- Mesos 版本 0.27.1。
- Docker 版本 1.6.2。
- Marathon 版本 0.15.3。

提示

部署 BrownField 微服务：

(1) 作为先决条件，必须在所有机器上安装 JRE 8。执行以下命令：

```
sudo apt-get -y install oracle-java8-installer
```

(2) 通过以下命令在指定为 Mesos 从设备的所有计算机上安装 Docker：

```
sudo apt-get install docker
```

(3) 打开终端窗口并执行以下命令，设置安装的库：

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv E56151BF
DISTRO=$(lsb_release -is | tr '[:upper:]' '[:lower:]')
CODENAME=$(lsb_release -cs)
# Add the repository
echo "deb http://repos.mesosphere.com/${DISTRO} ${CODENAME}
main" | \
sudo tee /etc/apt/sources.list.d/mesosphere.list
sudo apt-get -y update
```

(4) 安装 Mesos 和 Marathon，以及 Zookeeper 依赖：

```
sudo apt-get -y install mesos marathon
```

对 Mesos 从服务器的 3 个 EC2 实例重复上述步骤。下一步，在 Mesos 主服务器上配置 ZooKeeper 和 Mesos。

配置 ZooKeeper

在 172.31.54.69 机器上安装 Mesos 主服务器和 Marathon 调度程序。

在 ZooKeeper 中需要进行两个配置更改，如下所示：

(1) 第一步是将/etc/zookeeper/conf/myid 设置为 1~255 之间的唯一整数，如下所示：

```
Open vi /etc/zookeeper/conf/myid and set 1.
```

(2) 下一步是编辑/etc/zookeeper/conf/zoo.cfg：

```
# specify all zookeeper servers
# The first port is used by followers to connect to the leader
# The second one is used for leader election
server.1= 172.31.54.69:2888:3888
#server.2=zookeeper2:2888:3888
#server.3=zookeeper3:2888:3888
```

替换 IP 地址，只使用一个 ZooKeeper 服务器，但在生产场景中，需要多个 ZK 服务器来实现高可用性。

配置 Mesos

更改 Mesos 配置以指向 ZooKeeper，并通过以下步骤启用 Docker 支持：

- (1) 编辑/etc/mesos/zk, 将 Mesos 指向 ZooKeeper 实例用于选举:

```
zk://172.31.54.69:2181/mesos
```

- (2) 编辑/etc/mesos-master/quorum 文件, 并将值设置为 1, 在生产场景中, 需要设置为不少于 3 个:

```
vi /etc/mesos-master/quorum
```

- (3) 默认的 Mesos 安装不支持 Mesos Slave 服务器上的 Docker。为了启用 Docker, 更新以下 mesos-slave 配置:

```
echo 'docker,mesos' > /etc/mesos-slave/containerizers
```

运行 Mesos、Marathon 和 ZooKeeper 服务

执行所有配置更改, 启动 Mesos、Marathon 和 Zookeeper。

- 以下命令按顺序启动服务:

```
sudo service zookeeper start
sudo service mesos-master start
sudo service mesos-slave start
sudo service marathon start
```

- 在需要的时候, 可以使用以下命令停止这些服务:

```
sudo service zookeeper stop
sudo service mesos-master stop
sudo service mesos-slave stop
sudo service marathon stop
```

- 服务启动并运行后, 使用终端窗口来验证服务是否正在运行, 如图 9-12 所示。

```
ubuntu@ip-172-31-54-69:~$
ubuntu@ip-172-31-54-69:~$ ps -ef | grep zookeeper
ubuntu 1248 1180 0 16:33 pts/0 00:00:00 grep --color=auto zookeeper
zookeeper 17856 1 0 Apr10 ? 00:00:54 /usr/bin/java -cp /etc/zookeeper/conf:/usr/share/java/line.jar:/usr/share/java/log4j-1.2.jar:/usr/share/java/xercesImpl.jar:/usr/share/java/xmlParserAPIs.jar:/usr/share/java/netty.jar:/usr/share/java/slf4j-api.jar:/usr/share/java/slf4j-log4j12.jar:/usr/share/java/zookeeper.jar -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.local.only=false -Dzookeeper.log.dir=/var/log/zookeeper -Dzookeeper.root.logger=INFO,ROLLINGFILE org.apache.zookeeper.server.quorum.QuorumPeerMain /etc/zookeeper/conf/zoo.cfg
ubuntu@ip-172-31-54-69:~$
ubuntu@ip-172-31-54-69:~$ ps -ef | grep/sbin/mesos-master
ubuntu 1245 1180 0 16:33 pts/0 00:00:00 grep --color=auto/sbin/mesos-master
root 17866 1 0 Apr10 ? 00:00:28 /usr/sbin/mesos-master --zk=zk://172.31.54.69:2181/mesos --port=5050
--log_dir=/var/log/mesos --quorum=1 --work_dir=/var/lib/mesos
ubuntu@ip-172-31-54-69:~$
ubuntu@ip-172-31-54-69:~$
ubuntu@ip-172-31-54-69:~$ ps -ef | grep marathon.Main
ubuntu 1252 1180 0 16:33 pts/0 00:00:00 grep --color=auto marathon.Main
root 17217 1 0 Apr10 ? 00:04:07 java -Djava.library.path=/usr/local/lib:/usr/lib:/usr/lib64 -Djava.util.logging.SimpleFormatter.format=%25s%5s%6s%n -Xms512m -cp /usr/bin/marathon mesosphere.marathon.Main --zk zk://172.31.54.69:2181/marathon --master zk://172.31.54.69:2181/mesos
ubuntu@ip-172-31-54-69:~$
```

图 9-12

在命令行中运行 Mesos slave

在这个例子中，我们将使用命令行调用 Mesos Slave 服务器来显示附加的输入参数，而不是直接使用 Mesos Slave 服务器。停止 Mesos Slave 服务器，并重新启动：

```
$sudo service mesos-slave stop
$sudo /usr/sbin/mesos-slave --master=172.31.54.69:5050 --
log_dir=/var/log/mesos --work_dir=/var/lib/mesos --
containerizers=mesos,docker --resources="ports(*):[8000-9000,
31000-32000]"
```

所使用的命令行参数说明如下：

- `--master=172.31.54.69:5050`: 这个参数告诉 Mesos slave 连接到正确的 Mesos Master。所有 slave 都连接到 172.31.54.69:5050 这个 Master。
- `--containerizers=mesos,docker`: 支持 Docker 容器及 Mesos Slave 服务实例上的非容器的执行。
- `--resources="ports(*):[8000-9000, 31000-32000]"`: 此参数指示 slave 在绑定资源时可以提供两个端口范围。31000~32000 是默认范围。因为我们使用以 8000 开头的端口号，所以告诉 Mesos slave 允许暴露从 8000 开始的端口也很重要。

验证 Mesos 和 Marathon 的安装：

(1) 执行上面的命令，再在 3 个实例上启动 Mesos slave。因为 3 个实例都连接到同一主机，可以执行相同的命令。

(2) 如果 Mesos 从服务成功启动，控制台中将显示以下消息：

```
10411 18:11:39.684809 16665 slave.cpp:1030] Forwarding total oversubscribed
resources
```

上述消息指示 Mesos slave 开始定期向 Master 发送当前资源可用状态。

(3) 打开 <http://54.85.107.37:8080> 检查 Marathon 控制台。如图 9-13 所示，将 IP 地址替换为 EC2 实例的公有 IP 地址。

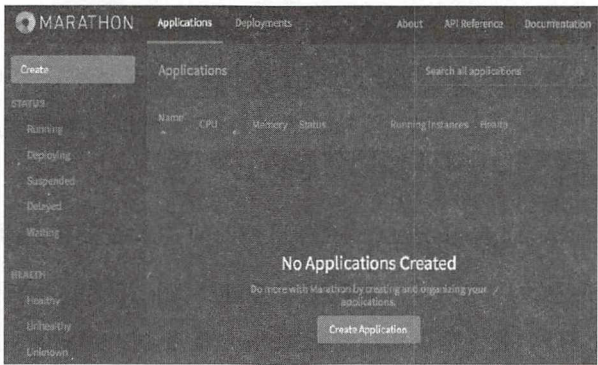


图 9-13

由于到目前为止还没有部署应用程序，因此控制台中的“应用程序”部分为空。

(4) 打开运行在端口 5050 上的 Mesos 控制台，转到 <http://54.85.107.37:5050>，如图 9-14 所示。

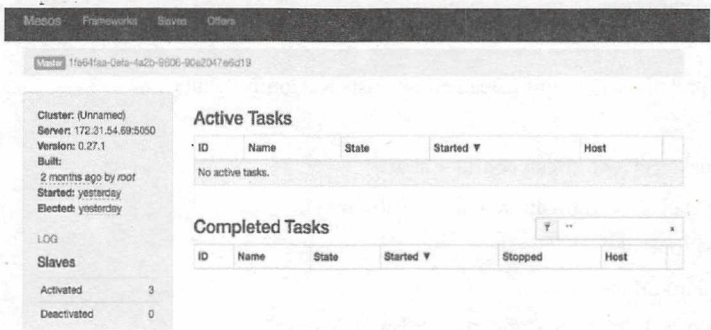


图 9-14

控制台的 slave 部分显示有 3 个激活的 Mesos Slave 服务可供执行，以及当前没有活动任务。

准备 BrownField PSS 服务

在上一节中，我们成功设置了 Mesos 和 Marathon。在本节中，我们将了解如何使用 Mesos 和 Marathon 部署 BrownField PSS 应用程序。

本章的完整源代码在代码文件中的第 9 章项目下提供。将 `chapter8.configserver`、`chapter8.eurekaserver`、`chapter8.search`、`chapter8.search-apigateway` 和 `chapter8.website`

复制到新的 STS 工作区中，并将其重命名为 chapter9.*。

(1) 在部署应用程序之前，我们必须在其中一个服务器中设置 Config 服务器、Eureka 服务器和 RabbitMQ。按照第 8 章“用 Docker 实现容器化微服务”中的“在 EC2 上运行 BrownField 服务”一节的步骤做。

(2) 更改所有 bootstrap.properties 文件以映射配置服务器 IP 地址。

(3) 在我们部署服务之前，需要对微服务进行特定的更改。当使用桥接模式运行 docker 化微服务时，我们需要告诉 Eureka 客户端要用于绑定的主机名。默认情况下，Eureka 使用实例 ID 进行注册，但是 Eureka 客户端是无法使用实例 ID 查找这些服务的。在上一章中应该使用 HOST 模式而不是桥接模式。

可以使用 eureka.instance.hostname 属性来完成主机名设置。但是，当在 AWS 上运行时，另一种方法是在微服务中定义一个 bean 来获取 AWS 特定的信息，如下所示：

```
@Configuration
class EurekaConfig {
    @Bean
    public EurekaInstanceConfigBean eurekaInstanceConfigBean()
    {
        EurekaInstanceConfigBean config = new
        EurekaInstanceConfigBean(new InetUtils(new
        InetUtilsProperties()));
        AmazonInfo info =
        AmazonInfo.Builder.newBuilder().autoBuild("eureka");
        config.setDataCenterInfo(info);
        info.getMetadata().put(AmazonInfo.MetadataKey.publicHostname.g
        etName(), info.get(AmazonInfo.MetadataKey.publicIpv4));
        config.setHostname(info.get(AmazonInfo.MetadataKey.localHostna
        me));
        config.setNonSecurePortEnabled(true);
        config.setNonSecurePort(PORT);
        config.getMetadataMap().put("instanceId",
        info.get(AmazonInfo.MetadataKey.localHostname));
        return config;
    }
}
```

上述代码根据 Netflix API 提供的 Amazon 主机信息来自定义 Eureka 服务器配置。代码使用专用 DNS 覆盖主机名和实例 ID。从 Config 服务器读取端口。另外，假设每个服务使用一个主机，以便端口号在多个部署中通用。另外，也可以在运行时动态读取端口绑定信息。

在所有微服务中运行上述代码。

(4) 使用 Maven 重新构建所有微服务，并将 Docker 镜像推送到 Docker Hub，具体命令如下所示。在其他服务重复相同的步骤。执行这些命令之前，需要切换工作目录。

```
docker build -t search-service:1.0 .
docker tag search-service:1.0 rajeshrv/search-service:1.0
docker push rajeshrv/search-service:1.0
docker build -t search-apigateway:1.0 .
docker tag search-apigateway:1.0 rajeshrv/searchapigateway:1.0
docker push rajeshrv/search-apigateway:1.0
docker build -t website:1.0 .
docker tag website:1.0 rajeshrv/website:1.0
docker push rajeshrv/website:1.0
```

部署 BrownField PSS 服务

Docker 镜像现在已经发布到 Docker Hub 上。下一步是部署和运行 BrownField PSS 服务。

- (1) 在专用实例上启动配置服务器、Eureka 服务器和 RabbitMQ。
- (2) 确保 Mesos 服务器和 Marathon 在配置为 Mesos Master 服务器的计算机上正常运行。
- (3) 如前所述，使用命令行在所有机器上运行 Mesos slave。
- (4) 此时 Mesos Marathon 集群已启动并运行。接着可以通过为每个服务创建一个 JSON 文件来完成部署，如图 9-15 所示。

```
{
  "id": "search-service-1.0",
  "cpus": 0.5,
  "mem": 256.0,
  "instances": 1,
  "container": {
    "docker": {
      "type": "DOCKER",
      "image": "rajeshrv/search-service:1.0",
      "network": "BRIDGE",
      "portMappings": [
        { "containerPort": 0, "hostPort": 8090 }
      ]
    }
  }
}
```

图 9-15

这些 JSON 代码将存储在 search.json 文件中。在其他服务中也创建 JSON 文件。JSON 结构解释如下：

- **Id:** 这是应用程序的唯一 ID，可以是一个逻辑名称。
- **cpus 和 mem:** 设置了此应用程序的资源约束。如果提供的资源不能满足该资源约束，Marathon 将拒绝来自 Mesos Master 服务器的资源提议。
- **instances:** 决定了此应用程序的启动实例数。在默认情况下，部署后立即启动一个实例。Marathon 会一直保持该实例数。
- **container:** 此参数告诉 Marathon 执行器使用 Docker 容器来执行。
- **image:** 告诉 Marathon 调度程序哪个 Docker 镜像来用于部署。本例中会从 Docker Hub 存储库 rajeshrv 中下载 searchservice: 1.0 镜像。
- **network:** 该值用于 Docker 运行时启动新 docker 容器时要使用的网络模式，可以是桥接或 HOST 模式。本例使用桥接模式。
- **portMappings:** 端口映射提供如何映射内部和外部端口的信息。本例中，主机端口设置为 8090，也就是 Marathon 执行程序在启动服务时使用 8090 端口。当容器端口设置为 0 时，将为该容器分配相同的主机端口。如果主机端口值为 0，Marathon 随机选择端口。

(5) 还可以使用 JSON 描述符进行其他运行状况检查，如图 9-16 所示：


```

"healthChecks": [
  {
    "protocol": "HTTP",
    "portIndex": 0,
    "path": "/admin/health",
    "gracePeriodSeconds": 100,
    "intervalSeconds": 30,
    "maxConsecutiveFailures": 5
  }
]

```

图 9-16

(6) 创建并保存此 JSON 代码后, 使用 Marathon REST API 将其部署到 Marathon:

```

curl -X POST http://54.85.107.37:8080/v2/apps -d @search.json
-H "Content-type: application/json"

```

对其他服务重复此步骤。

上一步将自动把 Docker 容器部署到 Mesos 集群并启动一个服务实例。

查看部署

其步骤如下:

(1) 打开 Marathon 控制台。如 9-17 所示, 所有 3 个应用程序都已部署并处于运行状态。

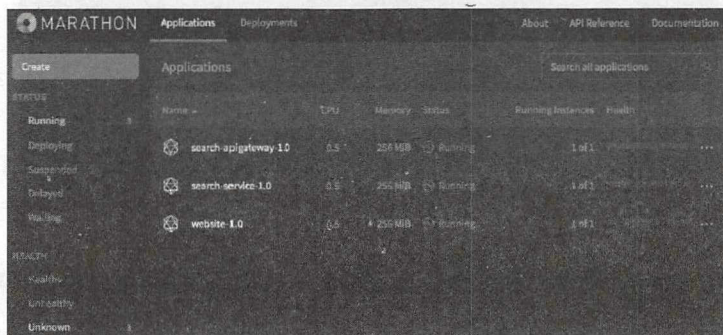


图 9-17

(2) 访问 Mesos 控制台。有 3 个活动任务, 它们都处于运行状态, 还显示了服务运行的主机, 如图 9-18 所示。

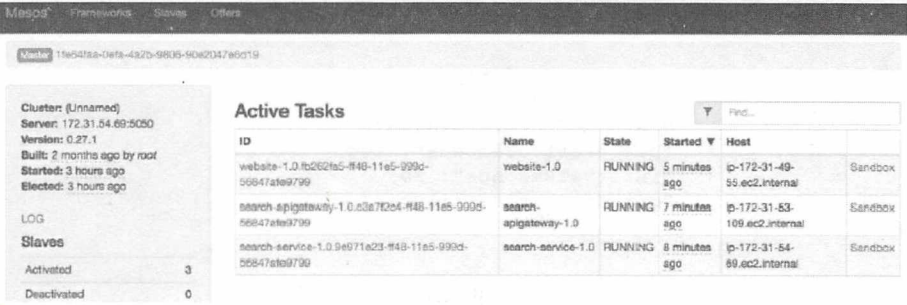


图 9-18

(3) 在 Marathon 控制台中，单击正在运行的应用程序。图 9-19 显示了 search-apigateway-1.0 应用程序的信息。在 “Instance” 选项卡中，可以看到服务绑定的 IP 地址和端口。

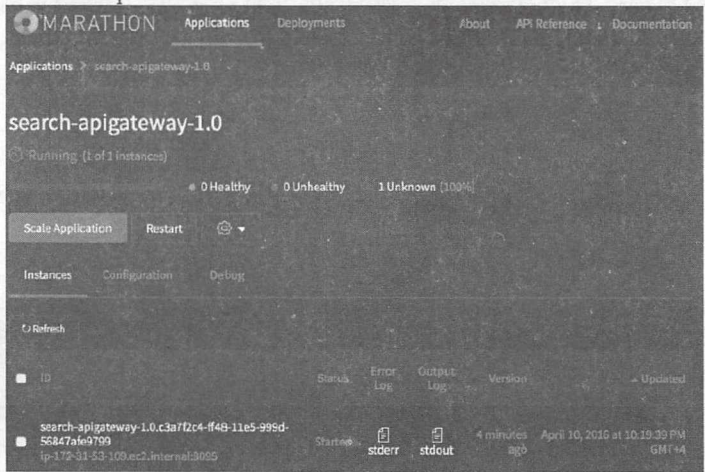


图 9-19

Scale Application 按钮允许管理员指定需要多少服务实例，可以用于扩容和缩容。

(4) 打开 Eureka 服务器控制台以查看服务是如何绑定的。注册服务时，会显示 AMI 和可用区。在浏览器中打开 <http://52.205.251.150:8761>，如图 9-20。

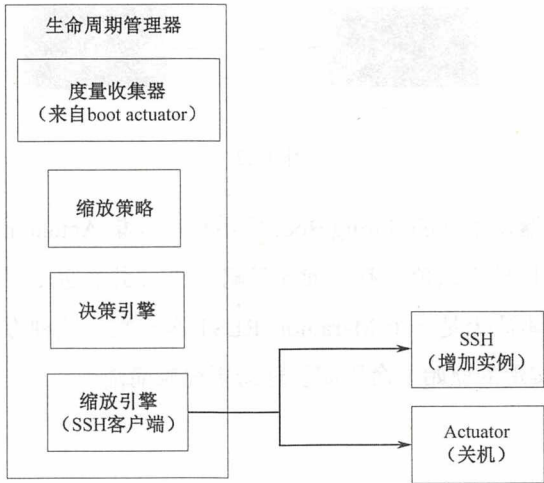
(5) 在浏览器中打开 <http://54.172.213.51:8001> 来查看 Website 应用程序。

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
SEARCH-APGATEWAY	ami-fce3c696 (1)	us-east-1b (1)	UP (1) - ip-172-31-53-109.ec2.internal
SEARCH-SERVICE	ami-fce3c696 (1)	us-east-1b (1)	UP (1) - ip-172-31-54-89.ec2.internal
TEST-CLIENT	ami-fce3c696 (1)	us-east-1b (1)	UP (1) - ip-172-31-49-95.ec2.internal

图 9-20

生命周期管理器的部署

第 6 章“自动化扩（缩）容微服务”中介绍的生命周期管理器具有根据需求自动伸缩实例的能力，还能够根据策略和约束来决定应用程序部署在何处以及部署方式。生命周期管理器的功能如图 9-21 所示。



Marathon 可以基于策略和约束来管理集群和部署集群。可以使用 Marathon 控制台更改实例数。

生命周期管理器和 Marathon 之间有冗余的能力。随着 Marathon 到位，不再需要 SSH 或机器级脚本。此外还可以把部署策略和约束委托给 Marathon。Marathon 公开的 REST API 也可用来启动伸缩。

用 Mesos 和 Marathon 重写生命周期管理器

我们需要一个自定义生命周期管理器，来从 Spring Boot Actuator 收集指标。如果伸缩规则超出了 CPU、内存和伸缩速率，自定义生命周期管理器也很方便。

图 9-22 显示了使用 Marathon 框架更新的生命周期管理器。

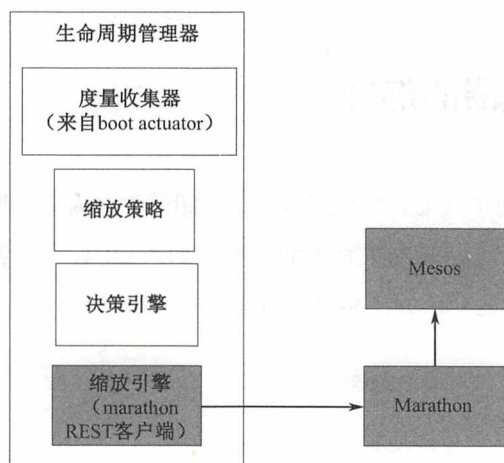


图 9-22

生命周期管理器从不同的 Spring Boot 应用程序收集 Actuator 的指标，将它们与其他指标结合，用以检查阈值。基于伸缩策略，决策引擎通知伸缩引擎进行扩容还是缩容。伸缩引擎本质上是一个 Marathon REST 客户端。这种方法比我们早期使用 SSH 和 Unix 脚本实现的原始生命周期管理器更优雅简洁。

技术元模型

我们已经使用 BrownField PSS 微服务覆盖了很多关于微服务的知识点。图 9-23 通过技术元模型将所有技术汇总在一起总结。

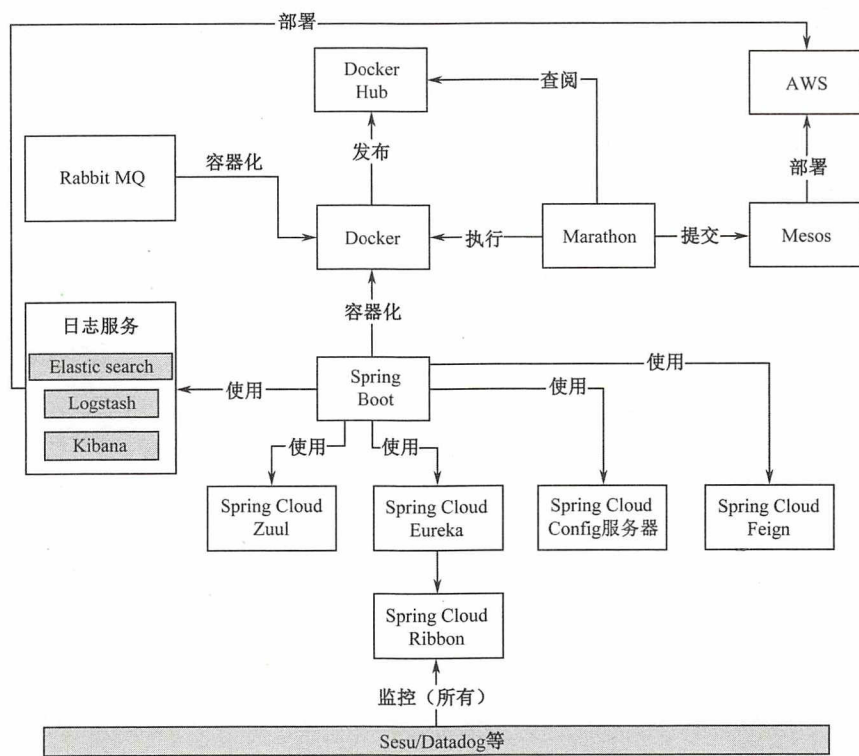


图 9-23

总结

在本章中，读者了解了集群管理和初始化系统管理大规模 docker 化微服务的重要性。

我们在深入到 Mesos 和 Marathon 之前探索了不同的集群控制和编译工具。我们还在 AWS 云环境中实现 Mesos 和 Marathon，以演示如何管理为 BrownField PSS 开发的 docker 化微服务。

在本章结尾，我们还探讨了生命周期管理器与 Mesos 和 Marathon 结合的位置。最后，我们使用基于 BrownField PSS 微服务实现的技术模型来做总结。

到目前为止，我们已经讨论了实现微服务所需的所有核心技术。



扫码答疑 共享资源



■ 译者简介

文彦峰，顺丰科技有限公司首席架构师，平台架构部、工程效率部和同城网络研发部负责人。主要从事业务架构、应用架构、平台架构、技术团队管理和研发过程管理，曾构建了黄金眼数据分析平台和腾讯云推送终端架构。

彭艳飞，曾在IBM公司从事研发工作，主要负责业务分析、系统架构、团队管理和研发过程管理，成功研发并上线运营的项目数十个。擅长架构高并发、自动扩缩容、报警自动化的分布式平台及Docker容器研发与调度等。

[PACKT]
PUBLISHING

策划编辑：董亚峰 dyf@phei.com.cn



责任编辑：刘小琳

封面设计：朝天世纪

ISBN 978-7-121-34085-7



9 787121 340857 >

定价：88.00元